SCM

Broadview

第2版

LOOKING AHEAD

未雨绸缪

理解软件配置管理

Understanding Software Configuration Management

董越◎著

電子工業出版社

未雨绸缪

理解软件配置管理

第2版

董越◎著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

软件配置管理为软件开发提供了基础性的支持环境，它与软件开发中的所有角色都有联系，因此本书是写给所有与软件开发有关的人士看的，而不仅仅是写给软件配置管理人员的。

本书分为两篇，上篇详细介绍了软件配置管理的基本知识；下篇主要讲述了软件配置管理的一些实际问题。本书的目标是让读者用很短的时间就能掌握软件配置管理的基础知识，在参与软件开发时，具备软件配置管理方面的基本素养。

本书的一大特色是作者用诙谐生动的语言来讲述专业的技术知识。文中充满睿智的调侃，为读者营造出一种轻松的氛围，让原本比较沉闷的技术阅读成为一种愉悦的享受！

未经许可，不得以任何方式复制或抄袭本书的任何部分。
版权所有，侵权必究。

图书在版编目（CIP）数据

未雨绸缪：理解软件配置管理 / 董越著. —2 版. —北京：电子工业出版社，2012.6
ISBN 978-7-121-16674-7

Ⅰ. ①未… Ⅱ. ①董… Ⅲ. ①软件－配置－管理Ⅳ. ①TP31

中国版本图书馆 CIP 数据核字（2012）第 056860 号

策划编辑：张春雨
责任编辑：付　睿
特约编辑：赵树刚
印　　刷：三河市鑫金马印装有限公司
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：720×1000　1/16　印张：17　字数：370 千字
印　　次：2012 年 6 月第 1 次印刷
印　　数：3000 册　　定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

第 2 版前言

本书第 1 版的写作完成于 2007 年秋天。四年以来，软件配置管理领域最大的发展变化是，以 Git 为代表的分布式版本控制工具日渐流行。本书第 2 版添加了专门的章节（3.5 节）介绍分布式版本控制工具，并在其他已有章节（第 2、3、4、5、7、12、16 章）中，介绍如何在不同的场景中使用这个新“装备”。

然而，引入分布式版本控制工具，只是第 2 版所修订内容的一小部分。读过第 1 版的读者在预览本版目录时会发现，本书的篇章结构发生了不少变化。比如，持续集成的内容进行了扩充，作为单独的一章（第 6 章）体现出来。再如，在讲解变更管理时，用一节的篇幅（10.2 节）讲解了 Scrum 等开发方法中，以特性（Feature）为单位的开发管理。

在具体内容的修改中，替换了第 1 版中两三成文字，以进一步突出重点、说清本质，并帮助读者更好地学以致用。此外，第 1 版中使用的若干术语在本版中进行了更新。比如改称任务单元为变更集，与业界通用术语相符。

本书第 2 版的具体内容改变了很多，但与第 1 版相比，本书的叙述风格和目标读者群没有发生什么变化。本书仍然力图以生动、易懂的叙述，让广大的软件开发人员、管理人员、质量保证与流程优化人员，当然也包括配置管理人员，对软件配置管理有一个基本的了解。

感谢电子工业出版社张春雨等编辑的辛苦工作，使本书第 2 版得以付梓。同时感谢如下评审者认真、细致的评审：我的配置管理同行李靖雯、李晓琴、李越、罗晓辉、王晓翔、赵延涛、郑博星，他们足以保证本书的专业性；从事软件研发中不同工作的关恒、齐文、张辉、庄孝德等同事，他们帮助我从不同视角审视考量本书；以及并不从事软件研发工作的我的夫人，她负责保证本书

的叙述通俗易懂，即使外行也能理解。感谢你们的帮助！

本书第 1 版在当当网（http://www.dangdang.com/）和中国互动出版网（http://www.china-pub.com/）得到了读者的较好评价，并且从出版至售罄，一直位列两网站软件配置管理书籍销量榜首。作为作者，最希望的事情就是写出的一点东西能得到读者的认可，对读者有所帮助。很高兴本书第 1 版初步实现了这一目标，并期望本书第 2 版能够给读者带来更大的收获。

董越

2011 年 10 月

推荐序

许多工科学生都参加过一门称之为“金工实习”的课程，会在生产车间内真实地体会车、铣、刨、磨、钻、铸、锻、焊等诸多生产工艺，同时也会领略到流水线生产的高效和有序。实际上配置管理就是保证这些工艺过程按生产大纲的安排，并行而有序地开展，最终将原材料或毛坯变为成品的关键方法。但软件毕竟和制造业产品或硬件不同，软件是“软”的，灵活性高，而且易变。因此，如何理解软件配置管理，并进而把握软件配置管理的内在规律，真正使软件配置管理起到为软件交付保驾护航的作用，的确是值得业界仔细思考和研究的一个课题。

从流程角度看，软件配置管理是整个软件开发生命周期中一个非常核心的管理过程。配置管理实际贯穿了从需求分析、架构设计、项目管理、开发、集成构建、测试，以及上线的全过程。这一过程不仅涉及宏观的项目进度控制、配置管理规范及计划、多地点开发规划等，也包括更细粒度的分支模型、构建及集成方式、变更处理流程，还包括微观的与开发人员直接相关的版本控制、差异比较与归并等。绝不夸张地讲，软件配置管理是一门“八面玲珑”的技术或艺术，而读者可以从董越先生的《未雨绸缪——理解软件配置管理》一书中切实感受到这一点。当开发人员从配置管理工具的版本树中静静欣赏自己作品的成形轨迹时，当集成人员从组件基线树把握整个应用组件的里程碑脉络时，当项目管理人员面对缺陷发展趋势以判断产品发布时机时，当构建生产人员顺利将正确版本的源代码从版本库中取出、构建并在生产系统投产时，配置管理及相关工具不仅仅是这些人员的有力助手，而且随着时间的推移它带给大家更多的将是一份信任和感动！

从人员角度看，软件配置管理人员在软件企业中起着非常重要的作用，绝

不是许多人头脑中的文档管理员、档案管理员或者什么其他的人员。他们是软件企业中所有软件资产的管理员，是最洞悉软件整个开发及变更过程并参与其设计的关键人员，是能与项目经理/人员、质量保证经理/人员、软件分析/设计/开发/构建/测试人员充分沟通并与之打成一片的“公关”人员，是了解操作系统管理、网络管理、各类开发语言和环境、深谙软件过程改进理论和相关工具的技术人员。通过本书，读者可以加深对配置管理的理解，加强企业配置管理的相关工作，重视配置管理人员的培养和提高。

全书尽管只有二百余页，但却深入浅出地道出了与配置变更管理相关的多个基本概念，如版本控制、配置管理、缺陷跟踪与变更管理、构建、组件、集成、分支等。同时书中根据不同实际场景给出了业界的最佳实践，而且处处渗透了作者在软件配置管理方面的心得，例如，对分支运用的分析、对组件复用的理解和实践、对于集成过程及集成责任的清晰阐述等。并且，本书结合工具（ClearCase、ClearQuest、Subversion、Bugzilla 等）简单明了地指出了当前流行的配置工具是如何实现这些最佳实践的。可以看出，《未雨绸缪——理解软件配置管理》一书是作者多年专注于软件配置管理理论和实践的结晶，书中不少观点实际上也代表了包括我在内的很多软件配置管理相关从业人员的感受和心声，是值得所有软件开发从业人员仔细阅读、细细品味及随时参考的好书！

非常感谢董越先生在紧张工作之余为大家所做的切切实实的贡献！

李纪华
2007 年 10 月

李纪华，现任 IBM 中国有限公司软件部 Rational 软件北方区技术负责人。曾经为国内外多家大型软件研发组织提供软件咨询服务，范围包括软件需求管理、软件配置与变更管理、IT 项目管理、实时系统测试等多个方面。此前在 Rational 澳洲亚太技术支持中心专门从事亚太地区 ClearCase 用户的技术支持工

作。在加入 Rational 及 IBM 之前，李纪华先后在 Platinum Technology 中国、Computer Associates 中国担任技术顾问，主要从事软件生命周期解决方案的咨询工作。在软件工程技术方面，李纪华有着多年的实践经验，目前主要专注于软件配置与变更管理、项目组合管理及企业应用现代化等领域。

（电子邮箱：lijim@cn.ibm.com）

前言

据说我小的时候，吃西瓜就喜欢吃尖儿。一桌西瓜刚切好，大人一不留神，就有很多块儿已经没尖儿了。这说明的第一个问题是，打小我就不是个老实本分的孩子。这说明的第二个问题是，西瓜数尖儿最好吃，最甜。如果肚子容量不大的话，应该先吃它。

我现在要做的事，就是和本书的读者们一起，去吃西瓜尖儿。软件配置管理（Software Configuration Management，SCM）可是个大西瓜，涉及很多方面，有很多细节在里面。广大读者，特别是并不以软件配置管理为生的读者，恐怕没这个闲心去钻研每一个方面的每一个细节，吃到西瓜露出青皮儿。这本书介绍的是软件配置管理的基本概念、基本思想、基本方法和基本要点。总之，是最核心的、框架性的内容。本书的目标是，让您用几小时的时间，掌握软件配置管理的基础知识。

本书是写给所有与软件研发有关的人士的，而不仅仅是写给软件配置管理专业人员的。原因之一在于软件配置管理的特殊性。软件配置管理与软件研发中的所有角色都有关系。开发人员集中精力于编写代码，但是需要版本控制来帮忙，以避免版本混乱，工作成果丢失；测试人员集中精力于测试，但是需要缺陷跟踪系统来帮忙，防止自己发现的缺陷开发人员忘了改，最后连自己都忘记了。项目经理关心项目进度，这同样和软件配置管理有关：如果能正确地运用分支这个手段，让不同的工作（比如系统测试和开发人员继续开发）并行起来，可以大大加快项目的进展速度。还有需求管理人员、系统架构师、用户文档编写人员、质量保证人员……每个人都要面对软件配置管理，都应学习使用它，并从它那里受益，因为软件配置管理为软件研发提供了基础性的支持环境。

另一个理由是，在很多情况下，软件配置管理应该怎么做，深深地受当时

特定情况的影响。而这特定的情况、特定的背景，往往是研发团队的其他成员要比软件配置管理工程师更清楚。就好像衣服穿多了还是穿少了，要听本人的意见，冷暖自知。

对于正在从事或打算从事软件配置管理工作的读者，这本书也是颇有价值的。日常的工作，可能会让我们只见树木，不见森林。而通过这本书，可以概览软件配置管理的全貌，系统地了解我们要做的事情，了解我们的使命。另一方面，本书所给出的软件配置管理的基本原理和基本方法，来自于业界的最佳实践和已经形成的解决模式。这些为解决特定研发背景下的具体问题，提供了很好的指导。我们需要做的是，考察具体问题、具体情况，融会贯通地运用软件配置管理的基础知识，产生优秀的具体解决方案，并付诸实施。

至于本书的书名，其中的“未雨绸缪”有如下三层含义。首先，记录历史，以供不时之需，这是软件配置管理的重要功能之一。其次，在软件研发中，随着软件研发规模的不断扩大，软件系统复杂性的不断增长，对软件配置管理的需求也会越来越强烈。因此，了解软件配置管理，是在为明天做准备。最后，本书帮助读者充实自己。谁不想成长呢？不论你现在专注于什么工作，不论你将来的理想是什么，只要与软件研发相关，相信本书都会有利于你的成长！

最后说一下本书的风格。就像这篇自序的风格一样，本书通俗易懂，偶尔还诙谐一下。希望读者在阅读时，能分享我写作时的愉悦。

致谢

平日里读书时，每当看到作者列出的长长的致谢名单，我心中总是不免起疑。但这种想法不会再出现了，因为我自己终于也经历了出书的过程——这需要许许多多的帮助和支持，这需要许许多多古道热肠的人。幸运的是，我遇到了他们。

首先感谢 SCMLife 的创始人孙常波先生。本书的写作，缘起于他的约稿。而在写作完成后，常波又付出了大量的心血，帮忙联系出版。没有你，就没有这本书！

传诚科技的晓黎女士，IBM 的浦雪（神仙姐姐）、王燕青、杨杨和朱宏，在联系出版的过程中，也搭手相帮。谢谢你们！

电子工业出版社的博文视点是本书的出版商。这本书能在博文视点出版，令我十分喜悦。感谢你们在阅读后把这本书归类为好书并决定出版，感谢你们几个月以来辛苦的工作。

本书推荐序的作者，IBM Rational 的李纪华先生，在软件配置管理领域有很高的威望。能得到他百忙之中的指点，进而欣然作序推荐，我感到非常的荣幸。

本书写作完成后，得到了如下同行和（前）同事的评审。

摩托罗拉: 石庆东、石莹莹、耿倩、张莹

爱立信: 赵延涛

诺基亚: 张晓昀

西门子: 雷琼

北电: 孙奕凌、朱东、周涛、陈胜利、刘晓钰、宋爱贤

雅虎: 刘振飞、徐晓明、李雪江

伟创力: 陆媛

新浪: 付丽

华为: 李靖雯

神州数码: 宋京

中兴通讯: 任艺

上海贝尔阿尔卡特: 唐雪蕾

是你们的反馈，帮助纠正了本书初稿中的某些技术问题，补充并完善了本书的内容；是你们的反馈，让我有了把本书公诸于众的信心。真诚地感谢你们！其中，孙奕凌女士的评审格外严谨、细致，甚至在遣词用句方面都帮忙做了很多润色。在这里特别感谢你，奕凌！

最后，感谢我的母亲，感谢您所给予我的鼓励和支持。在每章初稿完成时，您总是第一个读者，并随后提出中肯的建议。我没有辜负您的期望，在您的著述出版二十年后，我的著述也将要出版了。

目 录

上篇 基本话题 1

第 1 章 恼人不休的问题：什么是软件配置管理 2

1.1 问题的引出 3

1.2 有那么一类管理 4

1.3 为什么称做配置管理 6

1.4 其他一些比喻 8

第 2 章 基本的版本控制：记录历史并防止混乱 10

2.1 即使只有一个开发人员 11

2.2 建立公共存储区 12

2.3 防止版本覆盖 15

2.4 行话 18

第 3 章 当代版本控制方法 21

3.1 变更集的概念 22

3.2 以变更集为单位修改代码并提交 24

3.3 反方向：适时更新工作区 27

3.4 源代码的整体版本 29

3.5 分布式版本控制 32

第 4 章 集成：关注整体质量 35

4.1 集成的概念 36

4.2 保证提交的质量 38

4.3 狭义集成的步骤 40

4.4 在基线产生之后 44

4.5 质量保证：集成前、集成中、集成后 45

第 5 章 从源代码到运行中的程序 49

5.1 管理生成转换过程和相关资产 50

5.2 保证构建的可重复性 51
5.3 全量构建和增量构建 53
5.4 让构建更快 55
5.5 安装包的版本控制 57
5.6 管理测试运行环境 60
第 6 章 迈向持续集成 62
6.1 及早和经常地集成，持续集成 63
6.2 化繁为简 65
6.3 全程自动化 67
6.4 快！快！快！ 69
6.5 多层集成：当团队规模扩大以后 70
第 7 章 分支：减少等待，分头工作 74
7.1 文件级分支 75
7.2 产品级分支 76
7.3 典型应用：实现多层集成 79
7.4 温故而知新：实现隔离与共享 80
7.5 分支的其他典型应用 82
7.6 公共版本库中的分支 84
7.7 私有版本库中的分支 85
第 8 章 管理文档 88
8.1 文档的标识和存储 89
8.2 自带的说明信息 91
8.3 趋势：Wiki 93
8.4 趋势：数据文件和数据库 95
第 9 章 跟踪缺陷，直到消灭 98
9.1 别让发现的缺陷跑掉 99
9.2 状态转换图 100
9.3 准确记录，便于修复 104
9.4 消灭它？立刻消灭它？ 106

9.5 分析统计缺陷相关数据 109
9.6 关联缺陷记录与变更集 112
第 10 章 管理变更 114
10.1 管理增强 115
10.2 管理特性 116
10.3 瀑布模型中的变更 118
10.4 影响变更管理的因素 121
10.5 控制产品版本间的差异 123
10.6 表达产品版本间的差异 125
第 11 章 玄妙的学院派 127
11.1 配置识别 128
11.2 配置控制 130
11.3 配置状态报告 131
11.4 配置审计 133
11.5 一些软件研发模型中的配置管理 135
下篇 复杂场景中的配置管理 139
第 12 章 用分支实现交迭 140
12.1 补丁版本 141
12.2 复用另一条分支上的改动 142
12.3 甚至早在 1.0 版发布之前 145
12.4 甚至为了重要的内部版本 147
12.5 甚至为了每次集成 149
12.6 双分支结构 151
12.7 都是交迭 153
第 13 章 用分支实现变体 156
13.1 什么是变体 157
13.2 用分支支持变体 158
13.3 随主干继续演进 160
13.4 把公共改动放到主干上 163

13.5 变更管理对交迭和变体的支持 165
13.6 历史版本与变体版本的命名 166
第 14 章 用设置实现变体 169
14.1 在构建时读取设置信息 170
14.2 安装和运行时读取设置信息 171
14.3 资源与数据 172
14.4 何时使用哪种方法 173
14.5 设置对交迭的支持 174
第 15 章 用组件的组合实现变体 175
15.1 雏形：平台加应用 176
15.2 在不同阶段组装 177
15.3 复合基线 180
15.4 复合分支 183
15.5 实现软件复用：分支、设置、组件相结合 187
15.6 实现软件复用：架构、过程和组织 188
第 16 章 支持多地点开发 192
16.1 何时出现多地点开发 193
16.2 主要困难和解决方法 194
16.3 离线工作模式 195
16.4 远程访问模式 197
16.5 当地版本库模式 199
16.6 只读模式 201
16.7 多地点开发中的变更管理 202
第 17 章 支持跨组织合作 204
17.1 从开源到外包 205
17.2 管理第三方资产 206
17.3 本地记录第三方源代码的演进 207
17.4 再加上本地的改造 209
17.5 共同开发 211

第 18 章　管理软件部署 213
18.1　安装包的存储和管理 214
18.2　管理安装过程 216
18.3　记录运行系统配置 218
18.4　管理变更请求 219
18.5　管理网站 220
第 19 章　软件配置管理实施 223
19.1　不同阶段，不同挑战 224
19.2　如何完成一项改进 225
19.3　在一个项目的生命周期中 226
19.4　平衡集权与自治 231
19.5　管理众多的项目 234
19.6　软件配置管理团队的组织结构 235
结语　提倡务实的精神 238
附录 A　相关工具索引 241
参考书目 251
在线资源 252
书网互动 254

上篇　基本话题

1

第 1 章　恼人不休的问题：什么是软件配置管理

什么是软件配置管理？众说纷纭。这一章尝试用通俗易懂的方式，介绍软件配置管理的基本内容和基本目标，以便让读者有个初步的概念，有些整体上的把握。

1.1　问题的引出

软件配置管理工作者差不多都有这样的经验：在认识新朋友时，当别人问起自己所从事的职业，自然会回答道：“我从事软件配置管理工作。”接着，十有八九，会被问到下一个问题：“什么是软件配置管理？”总被问到相同的问题，倒还称不上是苦恼，真正的苦恼在于回答这个问题，因为软件配置管理真是不太容易说得清……解释了半天，结果往往是：“你这份工作好玄妙啊。隔行如隔山啊，我是搞不懂了。”

是的，软件配置管理确实不太好解释。软件开发过程中的其他工作，似乎都比它容易理解。开发人员在编写源代码；测试人员在测试、挑毛病；需求分析师配合用户确定需求，并且用准确的语言表达出来……虽然这样说未必严谨，但是至少能够得到一个大致的印象。然而，软件配置管理呢？软件配置管理是什么？

下面是软件配置管理的一个权威定义：

“A discipline applying technical and administrative direction and surveillance to identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements.”

“一套应用技术上和管理上的指导和监督的方法，用来识别和记录配置项的功能特征和物理特征；控制这些特征的变更；记录和报告变更的处理和执行的状态；以及验证其是否符合特定的需求。”[1]

如果你看得一头雾水，别担心，这不是你能力上的问题。大部分人和你的感受相同。这个定义，以及类似的权威定义，都高度抽象。用一两句话确实很

[1] 出自 IEEE-STD-610，并被 CMM、CMMI 引用。

难把握好软件配置管理这个概念，需要更多的描述才能把它说清楚。事实上，这一整本书，就是在认识和理解软件配置管理。而在这一章中，我们将用一些我们相对熟悉的概念来打比方、做对比，来讲解软件配置管理这个概念。通过这样一种方式，让大家对软件配置管理有一个虽然初步但比较正确的认识。

1.2 有那么一类管理

管理是个很宽泛的话题。大到管理企业、组织生产、治理河川、治国平天下，小到制定周末出游计划、整理通讯录、清理垃圾文件。在这里我们可不是要讨论这么多的事情。我们来重点看看下面这几个：

- 库房管理

库房管理员小张关心仓库里存放了哪些东西，分门别类放在哪儿，每样有多少个，何时存进来的，谁存进来的，何时提走，谁提走的（提走时要有出库单才行）。

- 财务会计

小王负责会计出纳工作，她要把公司的各项费用和各项收入分类记录在各种财务表格中，并进行统计。除了记录之外，她要确保合乎规定的票据才能在她这里报销。

- 资料保管

小李是总工程师办公室的一名秘书，她负责公司所有工程项目资料的统一保存。基本方法是，按照项目分门别类地保存；对每份资料编号；记录收录、出借、归还情况。

- 图书借阅

小赵在图书馆工作。在进新书时，她要把书籍编号，并分门别类放在合适的地方。更多的工作是书籍借阅和归还的管理，要记录借阅和归还的情况，还

要保证借阅和归还符合流程规定。

上面列举的管理工作，有一些共同特点。首先，它们都是要管理某种资产：可能是物资，可能是钱财，或者信息类的资产。

其次，它们都在做一些关于存放和记录的事情：分门别类存放在合适的地方；记录当前的情况，保存形成历史记录；记录“进进出出”的情况；做统计，做汇报，并在“用户”查询时提供数据。

最后，除了被动的记录外，还要做一些控制性的工作，把好关，按流程规定来，以免出乱子，损害了管理的对象。

软件配置管理也是这样。首先，软件配置管理是关于软件资产的管理。什么是软件资产呢？源代码、设计文档等文档、可以运行的程序、自动测试脚本、编译器等工具和环境……所有在软件研发过程中使用的或产生的，有价值的值得保存的东西，都是软件资产。软件配置管理就是关于这些内容的管理。

其次，软件配置管理要管理软件资产的存放和记录。把软件资产——主要是源代码，放在合适的目录结构里，放在合适的地方存储，防止丢失或者弄乱。还要记录谁“借”出了什么文件，什么时候“还”的。在这一“借”一“还”的过程中，如果开发人员修改了它，软件配置管理就要记录下这些修改。软件配置管理关心：这个文件的各个历史版本是否记录下来了，以便今后翻阅；各次修改的修改者、修改的原因是否记录下来了，以便将来可以理解当时的情形，理解为什么做出这样的改动。

最后，软件配置管理也要做一些控制性的工作。比如，两个人同时修改同一个文件的时候，可能会导致其中一个人的改动丢失，也就是常说的版本覆盖。要做一些控制性的工作防止这样的事情发生，可以让他们一个改完了另一个再改，也可以让他们同时改，将来再合并。再如，有的开发团队规定：只有通过同行评审和单元测试的源代码修改才能提交给集成工程师；集成工程师必须在

源代码通过编译链接和粗略测试时，才可以发版本。

这么看来，软件配置管理就属于“这一类”管理。当然，软件配置管理也有它自己的一些特点。比较重要的一点是，软件资产在不断变化，不断演进。项目初始的时候，可能只有一份简单的项目计划，而项目结束时，已是可以交付给用户的产品。如果缩小视野，单就某个源代码文件来看，也会看到，通常它会在项目的某个时刻，由某个开发人员创建第一个版本，然后，可能有不同的开发人员不断修改它，产生新的版本。软件配置管理就是围绕着这些事情展开管理的。

所以说，软件配置管理是关于不断演进的软件资产的管理，这主要包括两个方面：一是合理的存放和记录，二是对资产的变化加以流程上的控制。

1.3 为什么称做配置管理

机器由零部件组成。例如，汽车一般由底盘、发动机、车身和电器设备四大部分组成。其中，汽车底盘一般包括传动系、转向系、制动系和行驶系。传动系主要由离合器、变速器、传动轴和减速器等部件组成。再往下分析，基本就是零件了。

机器由正确型号的零部件配置而成。每个零件都有型号、编号，零件组成的部件也有，一直到整个机器，一辆汽车。要保证制造出来的机器是好用的，就要保证选取的零部件型号正确。那么，容易想到——应该有某种列表或文档，标明各零部件型号和组成关系，也就是说，要标明配置信息。而当配置有变动的时候，要更新这样的列表或文档。并且，这种变动不能随随便便，是否应该先让总工程师批准？是否应该做相应的测试？这些都属于对配置的管理。

从软件配置管理的视角看，软件也是这么配置出来的。往小了说，各个源代码文件的正确版本配置在一起，编译产生了正确的可运行的程序。往大了说，

若干软件组件的特定版本，配置构成了特定的软件产品。而有些软件组件，可能参与了不止一个软件产品的配置构成。而当某个软件组件参与不止一个软件产品的配置构成的时候，可能是这个软件组件的同一个版本，也可能是不同版本。看，问题有多复杂，不管理怎么行！

配置管理，管理系统的配置：硬件配置管理，管理硬件系统的配置；软件配置管理，管理软件系统的配置。注意这里的“系统”这个词。配置管理要管理每个零件，每个源代码文件，但管理的根本目的是保证它们能够组成一个可运行的系统。说得专业一点，那就是，配置管理关注系统的**集成**（Integration），或者说，关注系统的**完整性**（Integrity）。这是配置管理的一个重要特点。

此外，如何**复用**（Reuse）开发成果，让不同的版本、相似的产品之间共同的部分只开发一次，让众多项目能够多快好省地完成，也是配置管理关注的重点。这常常需要综合运用分支、设置、组件组合等方法。除了软件复用，多地点协同开发、与第三方的合作等，也都需要配置管理的支持。

做好软件配置管理，与做好硬件的配置管理相比，要更难一些。这是因为软件配置管理的特殊性。

第一，软件更容易发生变化，版本更多。一个开发人员，修改一个 Bug，可能半个小时就搞定了。于是，半个小时前与半个小时后，已经是不同的版本了。更何况，不止一个开发人员在工作。如此快速的、众多的变化，如果靠一个书记员手工记录相关信息，那恐怕比较累，所以需要某种自动化的工具，以提供这方面的支持。另一方面，在如此快速的、众多的变化中，还要保持系统的完整性，显然它的挑战也更大。

第二，软件的耦合性更高。当开发人员为某个任务改动源代码的时候，经常要改动不止一个文件。在目录结构上，这些文件可能相距遥远。组件/模块间的接口，往往并不像把鼠标线插到 USB 接口那么简单。某个模块的变化，常会

影响到相关模块[2]。这个特点，使得在软件领域，需要格外关心系统的完整性。要尽可能早、尽可能频繁地集成，保证产品作为整体是可以运行的。另一方面，一个模块、一个源文件可能被几个开发人员改动，他们可能出于不同的目的，改动不同的位置，甚至相同的位置。因此，版本更容易混乱或相互覆盖。

做好软件配置管理不容易。软件配置管理相关的工具，比如源代码版本控制工具等，可以提供支持，提供便利，同时避免出现问题。引入工具很重要，而用好工具，配合合适的流程和策略，以支持特定研发组织的具体研发场景，则更为重要。

1.4 其他一些比喻

- 保险柜

软件配置管理为软件开发提供了一个保险柜。保险柜里，存的都是值钱的东西。存进保险柜，是因为怕自己不小心弄丢，或者被偷走。软件资产也一样，甚至比金戒指之类更值钱。软件资产也会丢失，特别是源代码。比如，一个软件项目完成后，如果没有进行存储/归档等工作，等再过几个月，想基于版本 1.0 开发版本 2.0 的时候，可能会发现 1.0 的源代码找不着了……无奈，只好从头写。这是自己不小心弄丢的情况。软件资产还有可能被窃取或泄漏。虎视眈眈的竞争对手，无孔不入的商业间谍……所以，一定要把软件资产放进类似保险柜的地方。

- 岩钉

这是来自攀岩者的经验。系上保险绳，每向上攀一小段，就在岩壁上打个岩钉。这样，即使偶尔失手，也不会从半山坠到谷底，只是向下滑一小段。软件开发也是一样，适当保存历史版本，可以在失手的时候回退到上一个安全的

[2] 好的程序设计，特别是好的总体架构设计，有利于减少这样的耦合性。尽管如此，软件的耦合性仍是比较高的。

地方。这里的版本，不仅仅指某个具体文件的版本，也指整个产品的版本。不仅指源文件，也包括需求、设计、测试用例……当我们关心软件产品的部署和运行情况时，版本还意味着——某个软件，上次安装的是第几版，这次升级到哪个版本，如果升级失败，应该回退到哪一个版本。

- 脚印

“一步一个脚印”有两层含义。首先，先走好这一步，踩实了，踩稳了，再走下一步。软件研发也是这样，需要基线，需要里程碑，需要每个迭代结束时，向内部或外部的发布。这些是项目的脚印。在每个脚印处，我们要认真检查，是不是踩实踩稳了。这可能是通过相关人员的评审，领导的审批，可能是通过软件测试，也可能是通过某些检查。

其次，一个个的脚印，就构成了足迹。它告诉我们是如何一路走来的，走的是哪条路。必要的时候，我们可能会回顾。还有可能，我们会回到半路，以便从那里再闯一条新路出去。对应到软件开发，我们就是要保存历史上的版本，以备不时之需。

好了好了，据说所有的类比和比喻都是蹩脚的。软件配置管理是什么？软件配置管理就是软件配置管理。如果再多说一些，那就是，它是关于不断演进的软件资产的管理。这主要包括两个方面：合理的存放和记录，适当的控制；它特别关注软件系统的集成。让我们在随后的章节里，更细致地学习和研究吧！

2

第 2 章　基本的版本控制：记录历史并防止混乱

这一章的内容，是基本的版本控制。如果你接触过一些版本控制工具[1]，那么阅读这一章对你来说会很轻松。但是这仍然会是有趣的，值得一读的，因为你会从中了解在远古时代[2]人类是怎样一步一步发明了软件配置管理的。

[1] 一些版本控制工具也被称为软件配置管理工具。事实上，它们提供的功能也的确超越了版本控制。但是，在本书中，我们把以版本控制为主要功能的工具仍称为版本控制工具。而用软件配置管理工具来统称软件配置管理相关的工具。

[2] 是指软件配置管理的远古时代。大概 20 世纪 60 年代末 70 年代初，软件配置管理的概念开始提出。

2.1　即使只有一个开发人员

即使只有一个开发人员，也需要软件配置管理。只是，这时做软件配置管理的方法，主要是一般性常识，因此常常意识不到做了软件配置管理。让我们来看一看。

开发人员会经常随手按 Ctrl+S 键[3]，这一个动作的意思是告诉计算机，保存当前内容到磁盘。这样心里比较踏实。因为有无数潜在可能，会让尚未保存的修改化为乌有。比如邻桌把电脑电源线踢了，比如编辑器崩溃了，比如操作系统崩溃了，比如一只老鼠迅速跑过键盘，后面还跟着一只猫……

有经验的开发人员，会做得更多。他们会经常备份自己写的源代码。比如，在对某一个文件进行大规模修改之前，他有可能先把这个文件复制一下，并给这个备份文件取个名字，比如 protocol.c.bak。因为他知道，万一涂改得产品转不起来了，可以把文件当前版本和上次保存的版本做个比较，看看修改了哪些地方，到底是什么修改导致了问题。还有可能干脆就放弃最近的修改，从上次保存的版本开始，换一种思路写。

有些开发人员就懒一些。备份一个个文件太麻烦了，干脆把整个目录或者整个产品的源代码都备份。需要备份的时候，把整个产品的源代码复制一下，并且取个有意义名字。比如，对于 Messenger 这个软件，某大侠做过这样一个备份——messenger.0823.to_add_icon，意思是说，Messenger 这个软件在 8 月 23 号这天的一个备份，打算开始做增加图标的工作。这样做备份，比按文件备份省事儿多了，而且更有意义——备份的是整个产品，恢复的是整个产品。并且，能够知道每次保存版本的原因（见图 2-1）。

[3] 对于不同的编辑器，“保存”这一快捷方式所对应的按键组合可能是不同的。Ctrl+S 是比较常见的组合。

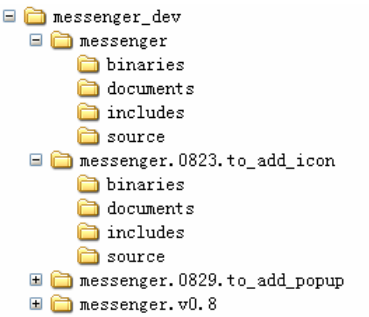


图 2-1　源代码整体备份

顺便提一下，懒是一种值得提倡的精神[4]。

也有的开发人员更谨慎。保存在本机，那万一磁盘坏了怎么办？万一磁盘被人卸了拿走了怎么办？应该再备份到另一块磁盘上，或者自己的另一台机器上，或者备份到服务器，备份到位于另一个城市的服务器，甚至备份到位于另一个国家的服务器……

看，这里所做的事情是，及时保存，及时备份，记录历史，以便追溯。这些都是软件配置管理的核心任务。当开发人数增多时，上述所有关心的内容依然存在，只是具体解决的方法可能会有差别。

2.2　建立公共存储区

当一个软件产品由多个开发人员共同开发的时候，面临的挑战又多了一些。为了让事情不要一下子变得太复杂，我们先假定，各开发人员有明确的“责任田”，即每个开发人员负责一个专门的模块，不存在两个开发人员修改同一处源代码的问题。好，基于这样的简化，我们来看看要面对的情况（见图 2-2）。

[4] 这是 Larry Wall 说的。作为 Perl 语言的发明人，他说，“A truly great computer programmer is lazy, impatient and full of hubris.”我懒得把这句话翻译成中文了。

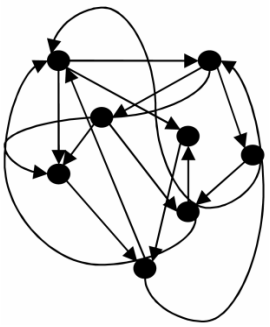


图 2-2　开发人员间随意传递源代码

假定每个开发人员有一台电脑，他和他的电脑是上图的一个点。每个开发人员都在编写程序。这涉及到两个基本问题：一是在修改程序之前，从哪儿拿到最新版本；二是在修改之后，把修改的结果提交到哪儿。

如果第一个问题得不到解决，开发人员就可能基于过时的程序开始自己的工作。假定，张大侠负责模块 A，王大侠负责模块 B，李大侠负责模块 C。模块 A、B、C 共同组成了软件产品。模块 A 向外提供的一个函数叫 Timer，有一个参数，标志着定时的时间。早先的时候，这个参数以毫秒为单位。最近，张大侠把它改成了以秒为单位。王大侠正在编写程序，在模块 B 里，他打算调用 Timer 这个函数。而王大侠手里的模块 A 的版本，是从李大侠那儿复制过来的。这个版本中模块 A 的 Timer 函数的参数，还是以毫秒为单位。所以，在王大侠的最新程序里，以参数 5000 调用 Timer 函数，让定时器停 5000 毫秒，也就是 5 秒。王大侠改完了程序，把模块 B 的最新版本交给了张大侠。张大侠很高兴，决定运行一下试试。后来……一根筋的张大侠在屏幕前枯坐了一个多小时[5]。

而如果第二个问题得不到解决，开发人员的工作就可能被湮没。假定还是刚才这拨人。上午，客户来找张大侠，说，现在有这么这么个 Bug，帮忙解决一下。张大侠三下五除二改好了，交给了客户。下午，客户找到王大侠，说，现

[5] 当然，这么一根筋的开发人员比较少见。但这件事情肯定会耗掉张大侠的一些时间，也许还有王大侠和李大侠的一些时间。

在有那么那么个 Bug，帮忙解决一下。王大侠三下五除二改好了，交给了客户。过了一会儿，客户发飙了，上午刚改好的 Bug，怎么又冒出来了！

爱思考的读者会发现，其实上面这两个问题是相互联系，相互影响的。

那么，我们怎么解决这个问题呢？解决之道是，将源代码流转的渠道从网状结构改成星形结构。也就是说，设立一个公共存储区，作为参照点和枢纽，大家不要再相互之间乱拷贝代码了，一概都从这个公共点取代码。等程序改完了，都把自己改的那部分全部传到公共存储区，别人再从那里取用（见图 2-3）。

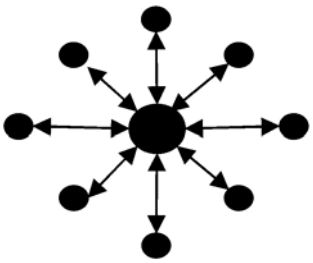


图 2-3　星形结构

来看看刚才的两个例子，用这样的方法是不是解决了问题。在第一个例子里，如果王大侠是从公共存储区里取得的代码，那么他取到的就是张大侠提交的，以秒为参数的 Timer 函数。王大侠就会在自己的程序里，用 5 而不是 5000 作参数来调用 Timer。问题解决了。[6]在第二个例子里，如果张大侠和王大侠改完代码后，不是直接发给客户，而是先上传到公共存储区。“用公共存储区里的代码编译生成”（因为“编译生成”不一定是在公共存储区里完成的，通常是复制到另外一个地方再“编译生成”）。有可运行的程序，再发给客户。这样，王大侠改完代码，发给客户的版本，一定既包含了张大侠的修改，也包含了王大侠自己的修改。问题也解决了。

[6] 严格地讲，是减少了问题发生的可能性。如果王大侠从公共存储区取代码的时候，张大侠还没有把改动放上去，那么还是会出现问题。在第 3.3 节、第 4 章和第 6 章，我们会继续深入讨论。

2.3　防止版本覆盖

在上一节里，我们做了一个假设，假定不存在两个开发人员修改同一处源代码的问题。遗憾的是，在现实世界里，这一假设常常是不成立的。经常发生的情况是，两个开发人员同时修改一处源代码。

假定，开发人员张大侠为修复 Bug 1 而修改模块 A、B，开发人员王大侠为修复 Bug 2 而修改模块 B、C。一开始，两个人都从公共存储区拿到了整套代码，即模块 A、B、C，并各自开始在其上工作。张大侠首先完成了 Bug 1 的修复。他把自己手上模块 A、B 的最新版本复制到了公共存储区。接着，王大侠完成了 Bug 2 的修复。他把自己手上模块 B、C 的最新版本复制到了公共存储区。灾难性的事情发生了。张大侠在模块 B 上的修改被覆盖了！

怎么避免这个问题？

不同的版本控制工具提供了不同的方法。但总的来讲，不外乎两种方法：通过监控，阻止同时修改这种危险事情的发生，即串行的方法；通过辅助，使同时修改的内容最终合并到一起，即并行的方法。

让我们来看看第一种方法：串行的方法。开发人员在修改前，先告诉版本控制工具，“我要修改它啦，帮我把它锁上，先不要让别人动了！”然后，从公共存储区拿到最新版本，修改，送回去。这时候再告诉版本控制工具，“我改好啦，把锁打开吧，别人可以动了！”（见图 2-4）。

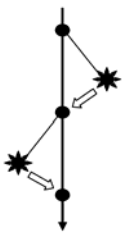


图 2-4　串行修改

显然，开发人员每次锁的范围小一点比较好。如果能锁一个文件，那就不要锁整个目录，更不要锁整个产品。不然，别的开发人员啥都不能碰了，啥都干不了了，只能干等着。不过，即使只锁定一个或几个文件，有时候还是会影响到别的开发人员，要是人家也要改相同的文件呢？

> **[Visual SourceSafe]** Visual SourceSafe，缩写为 VSS，是微软公司出品的一款“古老的”版本控制工具。它默认的方法就是第一种方法，串行的方法。在修改前，锁定相应的文件，以免别人同时修改。在修改后，解锁。通过设置，它也支持下面的第二种方法。它支持基本的版本控制功能，如版本保存、打标签等。另外，它还号称支持分支（见第 7 章）等高级功能。不过，不太好用。总之，它是一款初级的版本控制工具，并且微软已打算不再维护它。

第二种方法，并行的方法，可以解决这个问题。第二种方法允许不同的开发人员同时修改同一处地方，比如同一个模块，或同一个文件。在将来的某一时刻，把他们的修改合并[7]（见图 2-5）。

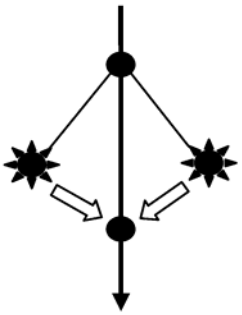


图 2-5　并行修改（省略了很多中间步骤和细节）

工具可以判断出何时需要合并，合并哪些内容。并且，在大多数情况下，工具可以自动完成合并。而在工具判定某些修改不能自动合并的时候，会提示人工合并。自动合并可靠么？从统计上看，这种自动合并尽管偶尔会出问题，

[7] 这一时刻，对于不同的版本控制工具，可能是不同的。有的主张在提交给公共存储区之前合并，没问题了，才可以提交；有的主张在提交给公共存储区之时合并，把它作为提交这一过程的一部分；还有的主张在提交给公共存储区之后合并……详情请见相关工具的说明文档。

但在绝大多数情况下都是正确的。很神奇吧！[8]

> **[Subversion]** Subversion，缩写为 SVN，默认支持第二种方法，也可以通过对文件加锁而支持第一种方法。SVN 是当前最受欢迎的开源的版本控制工具之一。它继承了它的前辈 CVS（Concurrent Versions System）几乎所有的优点，去掉了 CVS 几乎所有的弱点，还增加了很多新功能。对于中小型团队，它很值得考虑使用。而对于正在使用 VSS 和 CVS 的广大用户，它可以作为未来版本控制系统升级的目标。
>
> **[Git]** Git 支持第二种方法。Git 之父就是 Linux 之父 Linus Torvalds。2005 年，他为管理 Linux 内核的开发而研制了 Git。由于功能强、速度快，越来越多的项目特别是大型开源项目开始采用 Git，包括 Perl、Gnome、Android 等。Git 是分布式版本控制系统的典型代表。第 3.5 节将讲解分布式版本控制。
>
> **[ClearCase]** ClearCase，有时候缩写为 CC，同时支持两种方法。ClearCase 是 IBM Rational 出品的大型商用软件配置管理工具，其核心是版本控制。尽管能够听到对 ClearCase 的很多抱怨，比如昂贵、复杂、不好用，但它仍然是收费的版本控制软件中，市场份额最大的。它有两个版本：Base ClearCase 和 ClearCase UCM。
>
> Base ClearCase 向你提供的是文件、目录、版本、标签、分支、触发器和链接等“裸露”的环境。在此基础上，你可以进行比较自由地设置和二次开发，以满足你实际项目的需要。它的优点是灵活。
>
> ClearCase UCM 是开箱即用（Out of the Box）的，它提供了基于 Base ClearCase 的一套不错的封装。你只需要做一些简单的设置，就可以在实际项目中直接使用了。
>
> **[其他版本控制工具]** 市面上还有至少几十种版本控制工具，或者包含了版本控制功能的 ALM 工具[9]。除上文已给出的外，比较出名的还有 Perforce、StarTeam、AccuRev、TFS（Team Foundation Server）、RTC（Rational Team Concert）、Telelogic Synergy 等收费产品和 Mercurial 等开源免费工具。它们至少支持上文中两种避免版本覆盖的方法中的一种。工具很多，在本书中将重点介绍其中三个：Subversion、Git 和 ClearCase（特别是 ClearCase UCM）。注意，本书是不能代替相关工具的培训教程的。本书中关于相关工具的段落，主要是帮你把一般概念和原理，与具体某个工具联系上。

[8] 关于合并的理论和实践都很有意思，只是超越了这本书的使命。简而言之，版本合并是根据已知的三个版本，尽可能自动地求出第四个版本，即合并结果。在效果上，两个已知版本之间的差异，被加到第三个已知版本上，产生作为合并结果的第四个版本。

[9] ALM 的概念见第 9.6 节。

2.4 行话

直到现在，我们一直在说土话，什么公共存储区啊，下载源代码啊之类的。如果你用这些词汇跟人交流，那听起来有点不够专业，有损你的形象。其实更重要的是，“车同辙，书同文”[10]，我们得用一种公共语言说话。坦率地说，当前，在软件配置管理领域，术语名称并不统一，涵义也并不完全一致。不同的理论、标准和指南中，不同厂商的工具中，有不同的叫法和不同的内涵。这里我们尽力而为，介绍一些相对通用的。

我们上文所谈到的公共存储区，正式的名称是**版本库**（Repository），也叫存储库、存储池、配置库等。在版本库里，存储源代码的各个版本[11]。当然，版本控制工具通常不会傻乎乎地把文件的每个版本的全部内容都存上，而是只存储不同版本间有差异的部分，这叫**增量存储**（Incremental Storage）。

与位于中央服务器上的公共版本库相对应的，是每个人在自己的计算机上的私有的**工作区**，又称工作空间，英文原名是 Workspace。也有的时候叫工作目录树（Working Tree）、视图（View）、沙箱（Sandbox），等等。它是每个开发人员在版本控制之下工作的地方。开发人员从版本库拿到源代码，放在这里，然后在这里查看、修改、编译、运行和调试。等改好了，再把新版本的代码从这里送回到版本库。

> **[Subversion]** 将工作区叫做工作复制（Working Copy）。而 Repository 常被翻译为项目仓库。
>
> **[Git]** 将工作区叫做工作目录树（Working Tree）。版本库就是 Git 中的 Repository。
>
> **[ClearCase]** 将版本库叫做版本对象库（Versioned Object Base，VOB）。工作区大致对应于视图（View）这个概念。视图分为两种，静态视图（Snapshot View）和动态视图（Dynamic View）。静态视图是位于本机的一个目录树，这和其他工具很

[10] 请参阅秦朝相关史料。
[11] 严格地讲，不只是存储源代码，还可以存储其他内容，比如文档。

> 像。而动态视图有点像虚拟盘符，看着是在本机上，其实是连到了服务器上。事实上，它比虚拟盘符更玄妙，详情请参考 ClearCase 相关文档。

开发人员首先创建或者说初始化工作区，从版本库里把代码抓下来，填到工作区里。随后，开发人员在工作区里修改代码。修改完毕，把结果送回版本库，这被称做**提交**或交付（Deliver/Commit）。大家不断提交修改，导致版本库中的内容不断变化。如果某个开发人员想把版本库中新的内容同步到他已有的工作区里，那么他就要进行一个**更新**（Update）操作。

提交与更新。还有一套相似的词汇也常能听到：签入与签出。**签入**或检入（Check In，CI[12]）与提交的含义大体相同，是从工作区到版本库的同步。相应的，**签出**或检出（Check Out，CO）一般来说指的是从版本库到工作区的同步，大体对应于上一段所说的初始化工作区或更新操作。注意在不同的工具中，这些概念的含义不尽相同。

> **[Subversion]**使用签出（Check Out）、提交（Commit）、更新（Update）三个术语。在 Subversion 的世界里，签出意味着初始化工作区。
>
> **[Git]** 要更复杂一些。使用 pull 命令或 fetch、merge 等命令，代码从服务器上的版本库流到本地版本库，再流到工作区。在相反方向，综合使用 add、commit、push 等命令，工作区里的改动先流到本地版本库，再流到服务器上的版本库。这里提到的本地版本库的概念会在第 3.5 节详细讲。
>
> **[ClearCase]**在 ClearCase 的世界里，签出（Check Out）和签入（Check In）都是针对某个文件的。尽管工作区里已经塞满了从版本库下载的文件，但在着手修改某个文件前，先要以该文件名为参数调用签出命令，不得偷懒省略这一步。修改好了，再以该文件名为参数调用签入命令。而在 ClearCase UCM 里，还有 Delivery、Rebase 这一对儿词，我们随后再讲。

回顾一下前面讲过的星形结构（重画于图 2-6）。

---

[12] 根据上下文不同，缩写 CI 可能是指 Check In（签入）、Continuous Integration（持续集成，详见第 6 章）、Configuration Identification（配置识别，详见第 11.1 节）或 Configuration Item（配置项，详见第 11.1 节）。

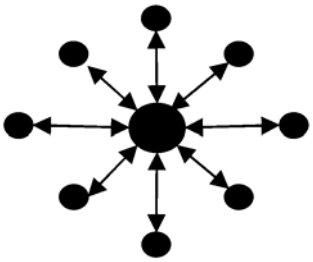


图 2-6　星形结构

中间那个大圆点是版本库，周围的小圆点是工作区。从大圆点到小圆点的代码流动是初始化和更新工作区，从小圆点到大圆点的代码流动是提交到版本库。

我们使用**版本控制工具**或称版本控制系统（Version Control System），来建立版本库、工作区，来进行提交、更新等操作。这比起手工的方式来，既便捷又安全。既然我们使用版本控制工具，那么就需要有人来负责工具的选择、安装、设置、二次开发、培训、疑难解答、制定相关规定和流程等一系列工作。这些是**软件配置管理工程师**（Software Configuration Management Engineer，SCM Engineer）的职责的一部分[13]。有时候，他们也被称为配置管理员，简称为配管（CM），也可能被尊称为配置经理（Configuration Manager）。嘿嘿，其实都差不多啦。

[13] 在职责划分比较细的公司里，这些工作可能并不完全是由软件配置管理工程师完成的。

3

第 3 章　当代版本控制方法

在几年前，“变更集”可是个时髦词汇。软件配置管理几大工具厂商争相宣布对它的支持。而如今的热门词汇是“分布式版本控制”。越来越多的软件开发团队在考虑迁移到分布式版本控制工具上。在这一章里，我们来看看，这些概念在本质上是什么，它究竟能带来什么好处，而工具能够提供什么样的支持。

3.1 变更集的概念

一般来讲，我们应该将一件事情做完，再交活儿。干到一半的工作，对别人意义不大，甚至有负面影响。编程序、改代码也一样。比如，张大侠这次本该修改三个文件，来添加一个功能。如果他刚修改了一个文件的时候，可能软件产品整体编译还未通过，就交上去，那就会出问题。当王大侠和李大侠拿到张大侠的代码的时候，他们也编译不过去。等找到原因，他们会对张大侠怒目而视，说不定结下江湖恩怨。所以说，我们应该在完成一项工作任务后，或者至少是取得了阶段性的工作成果后，再公布于众，把对各个文件的修改变更作为一个集合一股脑地提交上去。这是变更集这个思想的重要理由之一。

变更集（Change Set），又称变更包（Change Package），有时简称为变更（Change）[1]。一个变更集，对应着一项完成了的工作任务。变更集通常有一个标题，甚至更多的描述性文字，用来大致说明这个变更集完成的工作任务。这可能是修复一个 Bug，可能是小幅调整一个功能，也有可能是一项大的工作的分解，比如，为实现一个新特性（Feature），需要由几个开发人员协作完成，每个人分头完成自己的一个或几个变更集。总之，变更集通常对应于一个明确的工作任务，带有明确的目的，可以和其他变更集区分开。

从源代码实际变动的角度看，变更集是一个或多个文件上的一处或多处具体的代码改动的集合。也就是说，一个变更集涉及一个或多个文件，而每个文件的内容发生了变化，从修改前的版本变为修改后的版本（见图 3-1）。

[1] 当有人跟你说变更的时候，你要多加小心，不要想当然地认为他指的是变更集。变更的含义不只是变更集，更多的含义见第 10 章。

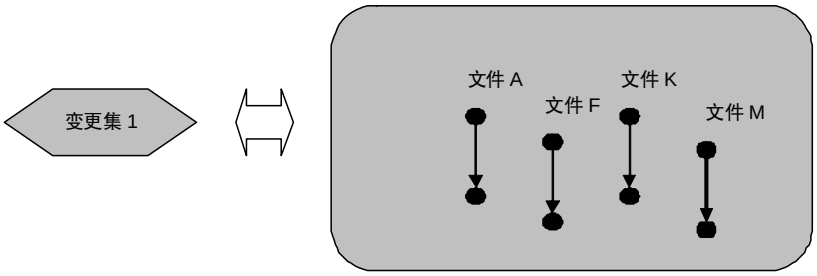


图 3-1　变更集

这样的一个一个变更集，累加在一起，实现了整个软件的从无到有，直到开发完成（见图 3-2）。

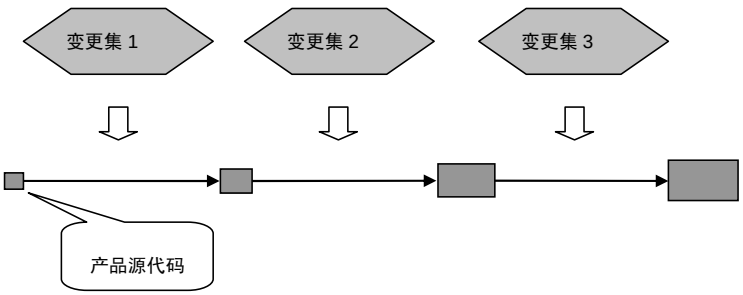


图 3-2　变更集的累加

用变更集来记录软件开发活动，便于明确做了哪些工作、谁完成的、什么时候完成的等团队协作必然遇到的问题。这样，便于交流，也便于管理。看一下张大侠这周完成的变更集的列表，喔，他真的是劳苦功高啊。

当开发人员把程序送交测试团队测试的时候，测试人员通常会问，这一版，你们改动了什么内容，增加了什么功能，修复了哪些 Bug？当程序发布给客户的时候，客户通常也会问相同的问题。如果是用变更集来记录软件开发的，那么，最近完成的变更集的列表，会对回答这类问题有很大的帮助。

而当我们在源代码改动的层面回顾工作的时候，同样能感受到变更集带来的好处。我们能够明确知道，某个 Bug 究竟是怎么修复的，某个新增加的功能，究竟对应于哪些文件的修改，修改了哪些内容。这样，如果怀疑以前的工作有疏漏，可以方便地追溯到过去的某个变更集——对代码的具体修改，看看是不是当时的改动有问题。如果你打算向编程高手学习，也一样——考察变更集各文件中代码内容的变化，可以体会高手编程时的良苦用心。

版本控制的重要目标是记录历史，提供参考，便于回退。变更集在这一点上也很有价值。相关的代码修改作为一个变更集，要么提交，要么不提交，要么回退，要么不回退，泾渭分明，绝不会出现中间状态，产生编译不通过之类的问题。

可以看出，项目管理人员、开发人员、测试人员和最终用户，都会从变更集中受益。

注意，每个变更集所对应的工作量，不要太大，它所对应的源代码改动，不要太多。否则，不好查阅修改情况，也不好管理[2]。如果要产生一个很大的变更集，那就要考虑把它划分成若干小一些的变更集，分别提交。另一方面，变更集的划分，也要避免没有必要的琐碎细节。这个火候如何把握？要从实践中不断获得经验。

3.2 以变更集为单位修改代码并提交

在本节中，我们来看一看开发人员是怎样使用版本控制工具的。当然，这里说的是支持变更集的版本控制工具。

在有些工具中，在改动代码之前，开发人员需要首先进行一个操作，创建一个变更集。也就是说，告诉版本控制工具，你要开始一个新变更集的开发了。

[2] 还有其他原因，我们会慢慢讲到。

这时候，你可以大致描述一下这个变更集对应的任务，版本控制工具会帮你记录。而其他不少版本控制工具不要求这一步，相关信息可以在完成一个变更集之后，提交的时候再补录。

一般来说，在完成代码改动后，提交代码改动到版本库。但有的时候，开发人员可能有这样的需求——能不能帮我保存个中间版本呢？也就是说，在改代码的过程中，在尚未完成任务时，能不能把阶段性的工作成果保存一下？一些版本控制工具提供了这样的功能，在安全的地方（比如版本库里的某个隐秘地方）**暂存**（Stage）它。这样，既保存了版本，又不至于干扰到别人，将来版本历史也显得齐整。类似的方法还有一些，在此不一一介绍。

在这个工作任务完成后，提交相关的修改到版本库。在进行提交操作时，可以撰写或补充该变更集的任务描述。

从下载版本库里的源代码，到提交修改，全过程如图 3-3 所示。

> **[Subversion]** 用相对简单的方法支持变更集。变更集有标题，这对应于变更集提交（Commit）时输入的注释。一个变更集，对应一次提交，也就是一次签入，不支持工作中文件的中间版本的保存。工作区中是一个未完成的变更集，完成后就提交。
>
> **[Git]** 与 Subversion 相比，有更多的缓冲。首先，从工作区正式提交（commit 命令）到本地私有版本库之前，可以先暂存（add 命令）。其次，可以在本地版本库里累积几个变更集，一起提交（push 命令）到服务器上的公共版本库，让别人看见。后者是分布式版本控制方法（第 3.5 节）带来的功能。当然，分布式版本控制方法也意味着，提交变复杂了，包括从工作区到本地私有版本库，再从本地私有版本库到服务器上的公共版本库这两步。
>
> **[ClearCase UCM]** 用相对复杂的方法支持变更集。在 ClearCase UCM 里，变更集大致对应于活动（Activity）。活动有标题，在活动创建时输入。而每个文件的修改又可以有注释。支持工作中文件中间版本的保存：工作区后面，对应着一个开发人员私有的流（Stream）。中间版本和最终版本都保存（checkin 命令）在私有流上，不会给别人捣乱。而把变更集从私有流提交（deliver 命令）到公共流后，大家就都

能看到啦[3]。总之，提交包括两步，从工作区到私有流，再从私有流到公共流。流这个概念，类似于分支，分支的概念在第 7 章讲。

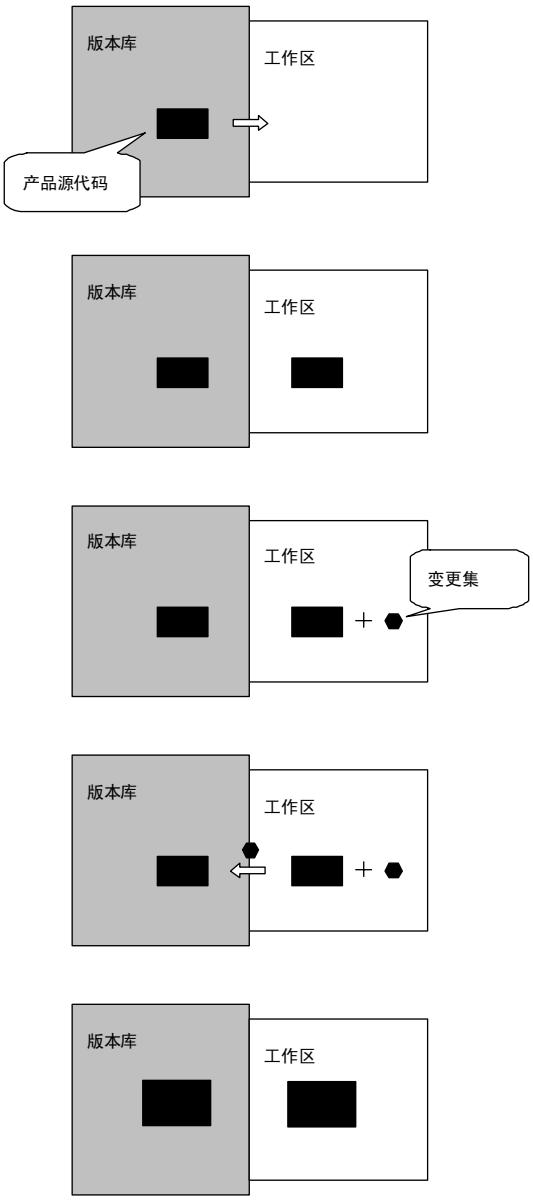


图 3-3　运用变更集

[3] 严格地讲，可能要等这个变更集进入到最新基线后，大家才能看到。基线的概念我们会在第 4 章讲到。

3.3　反方向：适时更新工作区

完成一个变更集的时间，可能很短，可能比较长。修复一个 Bug，可能几分钟就搞定了，而开发一个新功能，可能要几天的时间。在你埋头工作的时候，版本库可能已经发生了很大的变化。在自己工作于一个变更集的过程中，别人会不会向版本库提交了代码？如若这样，自己手上的代码会不会过时，不能和别人的最新代码共同工作了？自己的变更集开发持续时间越长，这样的危险越明显。还记得讲过的 Timer 的故事么？即使使用了版本控制工具，如果不及时进行更新操作，这样的事情同样可能发生。

为了保证你的工作成果能融入到当前的整个软件产品里，你需要适时更新你的工作区，得到版本库里的最新版本。让你的工作，你的改动，是在最新版本之上[4]（见图 3-4）。

> **[Subversion]** 更新就叫更新。
>
> **[Git]** 一个 pull 命令就可以完成这里所说的更新概念，但在原理上其实有三件事：从服务器上的远程版本库同步到本地版本库中的远程分支；从远程分支合并到本地分支；让工作区反映本地分支上的情况。
>
> **[ClearCase UCM]** 这里所说的更新，大致对应于 ClearCase UCM 里变基（Rebase）私有流，并相应地更新私有流对应的工作区。这里的变基是指用公共流的一个基线来更新私有流的起点，因此改变了工作区的基础。嗯，ClearCase 确实有点复杂。

更新可能会带来合并的情况，如果你所修改的代码本身，在版本库里已经有了别人提交的新版本。别担心，如上一章所述，这样的合并通常是自动化的，不会牵扯你太多精力。

那么，什么时候需要更新呢？首先，如上一节所述，在变更集开始之前，可能需要更新。以保证你是基于版本库里比较新的版本开始工作的，不要在开始的时候就落伍。这时候的更新，建立了你的关于这个变更集的初始工作环境。

[4] 严格地讲，可能不是更新到最新版本，而是更新到最新基线。第 4 章讲基线。

当然，如果你在开始新变更集之前刚做过更新，那就不是很有必要再更新一次了。

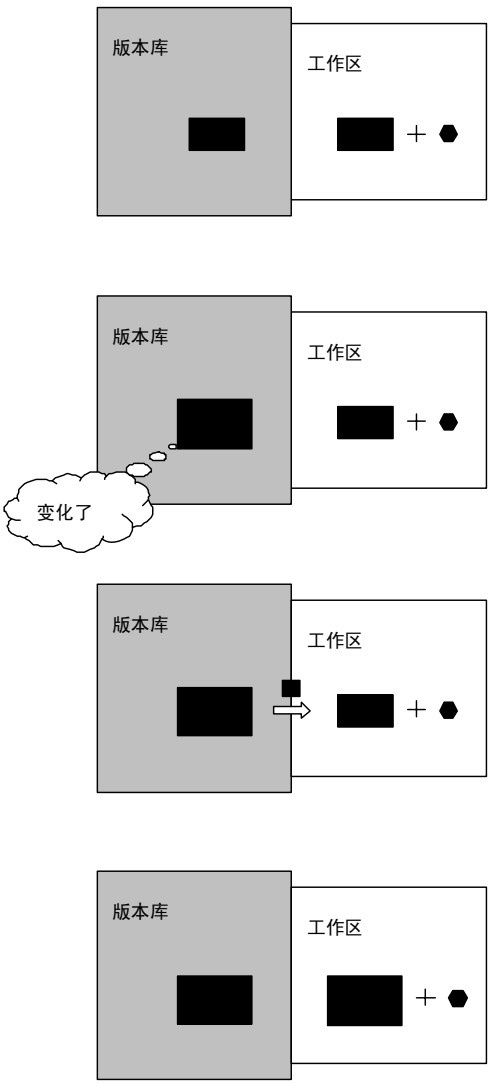


图 3-4　更新工作区

其次，在变更集完成的过程中，可能需要更新，以保证能跟上时代的步伐。

你埋头工作的时间越长，这样的需求就会越明显。当然，过于频繁的更新也没有必要，甚至有害。每次更新后，大概你需要重新编译，这可能很耗费时间。而且，版本库里的最新代码，难免有编译不通过，程序转不起来的时候。你不想每次都赶上吧？

最后，在变更集完成，即将提交的时候，最好做一次更新，并且测一下。以保证你新写的代码，是可以与别人的代码一同工作的，提交上去以后，是可以让其他人放心使用的。我们将在第 4 章里详细讲关于质量的事。另一方面，不少版本控制工具要求你更新到最新（或足够新）的版本之后才能提交，倒不是因为质量，而是因为工具本身的工作原理。

> **[Subversion]** 在 Subversion 的工作原理中经常需要提交前的更新。由于合并必须发生在工作区里，因此一旦有文件（可能）需要合并，就必须先更新再提交。
>
> **[Git]** 在 Git 的工作原理中也经常需要提交前的更新。在 Git 中，改动必须基于服务器上的远程版本库中的最新版本，否则无法从本地私有版本库提交到服务器上的公共版本库。
>
> **[ClearCase UCM]** 提交到公共流之前，不需要因为 ClearCase UCM 的工作原理本身的缘故而更新私有流及对应的工作区。因为除了开发人员自己的私有流有对应的工作区，公共流也有对应的工作区，提交到公共流时，可以在那儿完成代码合并工作。

3.4　源代码的整体版本

在很久以前，还没有变更集这个概念的时候，版本控制工具是以文件为单位进行管理的。第 2.4 节里介绍的签出和签入其实是那个时代遗留下来的概念。在那个时代，签出是告诉版本控制工具，我要修改某个文件啦。相对应的，签入是告诉版本控制工具，修改完成，请保存该文件的新版本。如今，签入演变成了提交，提交一个变更集到版本库。而签出演变成把所有的源代码，而非单个文件，从版本库里拿出来，填充工作区。

在那个以文件为单位进行管理的古老时代，每个文件有它的一个又一个版

本，形成版本演进的历史。然而要注意，不光每个文件需要保存版本，一个程序的所有的源代码作为整体，也要记录和保存版本，这些版本也形成版本演进的历史。因为大家关心：测试测的是哪个版本；交给用户的是哪个版本；这个版本测没测过；用户报告了一个 Bug，是指的哪个版本上的 Bug，等等。

那么，如何记录源代码作为整体的版本呢？在早期的版本控制工具中，是这么一个思路：若干文件组成了产品，每个文件的特定版本组合在一起，是产品的特定版本。所以，记录产品的某个版本，只需要记录这个版本，是由哪些文件、具体哪个版本组成的。在操作上，可以在相关文件的特定版本上，打个**标签**（Label / Tag），标签的名字，就是整体版本的名字。这样，将来搜索所有的以这个整体版本命名的标签，就能找到这个整体版本对应的所有源文件的正确版本，也就找到了这个整体版本对应的全部源代码（见图 3-5）。

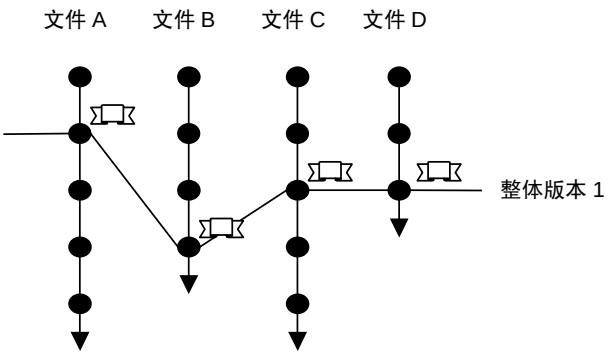


图 3-5　文件版本组成整体版本

这招不错，解决了记录整体版本的问题。但它也有一些弱点。比如，在每个文件上都要打标签，创建一个包含众多文件的程序的整体版本就可能会比较费时间。另一方面，不同整体版本的标签很可能打在某个文件的相同的版本上。这个版本上，可能挂着几个标签，可能挂着几十个标签，也可能挂着几百个。

当代版本控制工具不再使用上述方法记录源代码整体版本[5]。它们用的是另外一套思路，从另外一个视角看问题。还记得讲变更集的时候我们用的图吗？（重画于图 3-6）

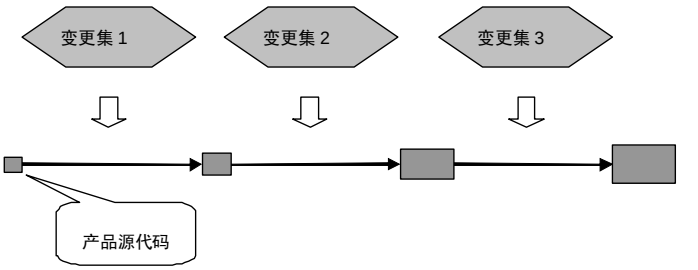


图 3-6　变更集的累加

每个变更集提交到版本库后，自然而然地形成了产品源代码的新的整体版本。一个又一个变更集提交上来，产品向前不断演进，或者说，不断形成新的整体版本。

版本控制工具通常会给这样自然形成的整体版本编号，比如 1、2、3……8573、8574、8575……或者更神奇一点，49A36B2、E331D55、8B52E13、94CC76F……

好像这些编号没有什么实际含义，不太好记……想好记一点也简单：给某个自然形成的整体版本本身打个标签，或者说起个别名（Alias）。比如，告诉版本控制工具，49A36B2 这个版本也叫 Rel2.3。

遥想当年，可能需要在成百上千个文件上打标签，可能耗时数十分钟或更长。如今，只需要打一个标签，通常秒杀[6]。

[5] 对于一些版本控制工具，准确地说，是看起来不再使用上述方法。另外，对于所有的版本控制工具，“整体的版本由各文件的特定版本组成”这个陈述总是成立的，不论是否用标签来标识各文件的特定版本。

[6] 如果你真的不知道秒杀是什么意思，那就姑且把它理解成“一秒钟之内完成”吧。另外，多关心足球篮球赛事、去玩玩网游、去淘宝上买点东西，都有可能促进对秒杀的理解。

> **[Subversion]** 用自然数列来为自然形成的整体版本编号。如果想有一个有意义的名字，那就用 copy 命令，把某个产品对应的源代码从版本库中的一个目录下复制到另一个比如叫 Rel2.3 的目录下。当然，这里说的复制，并不是真的复制，是逻辑上的，类似于链接（Link），因此是秒杀。
>
> **[Git]** 用一个比较长的随机数（被称做 SHA1）来为自然形成的整体版本编号。如果想有一个有意义的名字，那就贴个标签（Tag）。没错，秒杀。
>
> **[ClearCase]** Base ClearCase 是以文件为单位进行版本管理的。因此打标签可能会成为一个费时间的工作。当然，使用一些技巧，比如增量地打标签（只在自上次整体版本以来有变化的文件上打标签），可以改善性能，但同时也增加了复杂性。ClearCase UCM 是基于 Base ClearCase 的封装，它使用类似的技巧来改善性能，常能在几秒钟之内完成打标签的工作。

在这样的解决方案下，单个文件的版本怎么表达呢？当你询问版本控制工具的时候，它会告诉你，该文件在整体版本 8573 中被修改了，相关的变更集里记录着，这是为了不仅显示 GIF 图片而且显示 JPG 图片。后来该文件在整体版本 37420 中又被修改了，相关的变更集里记录着，这是为了把刷新频率提高一倍。

3.5 分布式版本控制

还记得我们讲过的星形结构（重画于图 3-7）那张图吗？

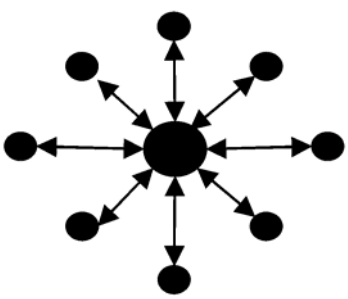


图 3-7 星形结构

在前面的描述里，我们给出了这样一种方式：星形结构中心那个点，是版本库，存储着源代码的所有版本，放在服务器上。周围的那些点，是工作区，里面塞的是代码的某个历史断面及正在进行的少量修改，存储在每个开发人员的计算机上。从周围到中间，是提交；从中间到周围，是更新。这是典型的集中式版本控制工具的工作方式。

然而，历史的车轮滚滚向前，一种新的工作方式崭露头角：**分布式版本控制**（Distributed Version Control）方式。在这种新的方式中，在星形结构中心那个点，仍然是一个版本库公共的版本库，存储在服务器上。而星形结构周边的那些点，在每个开发人员的计算机上，不再仅仅是工作区，即代码的某个历史断面，而是还有一个完整的私有版本库。这个私有版本库的内容与公共版本库存储的内容几乎一样。在开发人员看来，这个私有版本库是本地版本库，而服务器上的公共版本库是远程版本库。

开发人员在绝大多数时间里，只和本地的私有版本库打交道：把私有版本库中的代码拿到本地的工作区，修改代码后，放到私有版本库保存。查看代码历史，查看各种信息时，基本上只与本地私有版本库打交道。而在需要与其他人交换信息时，就在服务器上的公共版本库和本地的私有版本库之间进行同步操作：从公共版本库到私有版本库，或者反方向。

这种新方式的优点在哪里呢？主要有两个优点：首先，对个人外出工作、工程团队在现场修改源代码、项目研发分布在多个地点等多地点开发场景有很好的支持。类似于在一个站点内部，每个程序员可以有自己的私有版本库，与公共版本库相互独立并相互同步，不同站点可以有该站点自己的公共版本库，这些公共版本库之间相互独立并相互同步。此外，还可以考虑私有版本库与另一个站点的公共版本库直接同步。第 16 章将会详细介绍多地点开发的解决方案，分布式版本控制工具对此有很好的支持。

这种新方式的第二个优点是，版本控制工具具有更好的性能，可以支持更大规模的研发。在集中方式下，大量的操作都需要与公共版本库打交道，它的负荷很重，性能容易出现问题。而在分布方式下，只有很少操作需要与公共版本库打交道，通信量不大，因此开发人员会觉得版本控制工具的性能很好。

分布式版本控制还有很多优点。比如，由于每个开发人员都有自己的版本库，他就有机会改写历史，把本地版本库中的历史记录整理得清楚些，好看些，再同步到公共版本库。限于篇幅，这里不再细讲。

对分布式版本控制了解不多的读者可能会对它有些疑虑：每个开发人员都有一个本地版本库，那么会不会很占磁盘空间？每个开发人员都有一个版本库，那么权限是不是很难控制，源代码容易外泄？

对于第一个问题，由于采用增量存储和压缩算法，版本库的尺寸可能比工作区还小。对于第二个问题，如果“坏人”没有公共版本库的读权限，他就无法基于公共版本库建立本地版本库并获得源代码，就好像在集中式版本控制工具中，如果他没有公共版本库的读权限，就无法基于公共版本库建立本地工作区并获得源代码。而如果他有公共版本库的读权限，那么无论是分布式版本控制工具还是集中式版本控制工具，他都能取得源代码。因此，集中式版本控制工具并不比分布式版本控制工具更安全。

> **[Git]** Git 是分布式版本控制系统的典型代表。在拥有众多的优点的同时，大概唯一的弱点是，有点复杂，需要多花一些时间来学习。

在上一章的基础上，本章介绍了当代版本控制工具中的三个常见重要特性：变更集（第 3.1 至 3.3 节）、整体版本（第 3.4 节）、分布式版本控制（第 3.5 节）。当然，版本控制工具的功能不止这些。比如分支，就还没有介绍。第 7 章将介绍分支概念和典型用法。

4

第 4 章　集成：关注整体质量

集成是一个在各个行业中广泛使用的词汇。它给人的印象是：把不同的部分攒到一起，并且能够工作。在软件开发中，是不是也是这样呢？

4.1 集成的概念

集成最初的含义是，把小块的东西拼起来。如果更精确一点，那就是，不仅要拼起来，还要拼好。如果放在软件开发这个领域，那就是要让整个软件跑起来。张大侠说，他那个模块开发完了，能跑了。王大侠说，他那个模块也开发完了，能跑了。李大侠也这么说。你可不要以为万事大吉了，这几个模块放在一起，指不定啥样子呢。说不定张大侠和王大侠可能对两个模块之间的接口的理解不同，以至于这两个模块不能配合工作。也许李大侠没有考虑到王大侠的模块可能会给他传来一个超出范围的值，使得李大侠的模块在毫无征兆的情况下崩溃……

集成（Integration）的基本使命是，把产品的各个部分捏在一起，并且保证，产品作为整体是可以运转的，而不仅是每个模块、每个单元能在特定的开发调试环境、特定的数据和参数下可以运转。

这是从一个视角看。从另一个视角看，集成是把不同的人、不同的团队的改动捏到一起，并保证系统仍能工作。集成的不是模块，而是修改变化。每个人都在这个软件产品上工作，添加、修改、删除。每个变更集，可能在一个模块上修改，也可能涉及多个模块。每个人完成一个或好几个变更集。要保证这些改动是可以累加在一起的。也就是说，新完成的若干变更集，是可以合并在一起，可以编译通过，可以运行的。从这个视角看，不再是把产品的各个模块捏到一起，而是把对产品的改变合并在一起，基于已有版本，产生新版本。所集成的，是修改变化，是变更集（见图 4-1）。

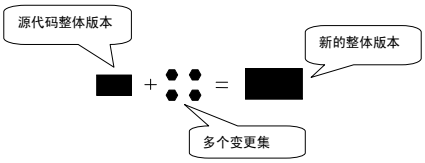


图 4-1 集成的含义

注意，即便是每个改动都是正确的，也不能保证捏到一起仍然是正确的。举个例子：假定有个应用程序，它的窗口的固定宽度是 600 像素，窗口里填充了背景图案，背景图案的宽度是 200 像素。也就是说，在横向上，该背景图案重复三次，挺好看。这时，因为一些原因，张大侠把应用程序的窗口的固定宽度从 600 像素缩小到 400 像素。而同时，王大侠换了张背景图案，背景图案的宽度从 200 像素换成 150 像素。这两个改动单独发生时，都没什么问题，窗口里背景图案还是齐边儿齐沿儿的。但是当这两个改动捏到一起以后，窗口里背景图案不再是齐边儿齐沿儿的了。

所以说，在软件开发领域里，集成有两个方面的含义：把各个部分捏在一起，并且保证可以工作；把各个改动捏在一起，并且保证可以工作。

注意集成有广义和狭义之分。上面是广义的集成的定义。照此定义，当开发人员在提交之前更新工作区，并作自测试的时候，他是在看他的改动是否能跟别人新近的改动一起工作。这是在做集成。当他做完他所在的模块的单元测试，又测了一下应用程序外在的行为的时候，他是在看他的模块是否能跟别的模块在一起工作。这也是在做集成。随后，系统测试团队对应用程序的一些版本做更加全面的测试，并把发现的 Bug 反馈给开发人员，以便修复，这个过程当然也是在做集成。等最终做好了集成，就正式发布给客户，上市卖钱。

而当人们谈论集成的时候，常常谈论的是狭义的集成，也就是集成工程师所做的工作。简单地说，集成工程师取源代码的最新版本，编译构建，并作粗略测试。如果在此过程中遇到了问题，就解决问题。都没问题了，宣布系统新的版本，供系统测试团队深入测试，也供开发人员使用。

广义的集成几乎是必需的。通过广义的集成，系统的各个部分得以配合工作，不同人的修改得以配合工作。而狭义的集成，也就是集成工程师做的集成工作，对于开发团队来说却不一定是必需的，或者说，可能很不起眼。这种情况常常发生在规模不足 10 人的小型团队里。在小型团队里，如果开发人员都能

保证提交的质量，那么版本库里的最新版本有严重质量问题的几率就很小，开发人员可以相当有把握地随时用公共版本库里的最新版本更新自己的工作区，不必担心程序因此而转不起来。我们先从这种情况讨论起。

4.2 保证提交的质量

开发人员完成一个变更集，然后提交该变更集。在提交之前，开发人员要做一些质量保证工作。原因很简单，如果张大侠提交了很烂的代码，导致王大侠和李大侠在更新他们的工作区后，程序跑不起来了，那么王大侠和李大侠就会围殴张大侠。

当然，文明社会，围殴事件很少发生。但是显然，张大侠的不负责任的行为，对团队其他的成员们造成了伤害，耽误了他们的宝贵时间，往大里说，耽误了项目进展。为此，张大侠应做检讨，改过自新。如何改过自新呢？

在前面的章节里，我们已经提到了一些。首先，要在一个变更集完成之后再提交。干到半截的工作，可能会给团队其他成员带来困扰（详见第 3.1 节）。其次，要适时更新工作区，尽力避免因为跟大家不同步而带来的问题（详见第 3.3 节）。下面是其他一些方面。

在提交前，应该做充分的自测试。首先是**单元测试**（Unit Test）。尽可能保证自己所编写的那一部分是没有问题的。进而，对系统整体进行测试，保证自己所编写的那一部分，与软件的其他部分的配合是没有问题的。这些测试都可以考虑自动化。

在这个敏捷（Agile）时代，谈到单元测试，就不得不提及**测试驱动开发**（Test-Driven Development，TDD）。简单地讲，在这种方法中，是先写好自动测试脚本，再开发，直到测试运行通过。这样做的好处是，用简单的脚本而非繁复的文档，在开发前明确开发的目标，而在开发后又能自动地检验该目标完成的情况。

除了测试外，有的团队还会引入**同行评审**（Peer Review），又称做代码走查（Code Inspection）等。在开发人员完成一个变更集中的代码修改后，把相关的修改，送交其他开发人员（通常是 1 到 4 个）仔细阅读，找出并排除错误。同行评审应该得到工具的一定程度上的支持，让其他开发人员能够方便地看到要评审的内容：这包括，变更集中各文件修改后的版本，和各文件的改动本身，即修改前和修改后的差异对比。如果工具能够方便地记录评审意见，那就更好啦。

> **[Gerrit]** Gerrit 是一款开源的同行评审工具，与 Git 相集成。它“拦截”开发人员提交的变更集，只有通过了同行评审，才会“放行”。Gerrit 通过网页展现变更集；评审人员可以把评审意见写到对应的代码行附近，方便直观。

与同行评审相关的是有些敏捷开发团队采用**结对编程**（Pair Programming）技术，目的之一也是提高程序质量。结对编程技术是一个非常简单和直观的概念：两位开发人员肩并肩地坐在同一台电脑前合作完成同一个设计、同一个算法、同一段代码或同一组测试。与同行评审相对比，同行评审是在变更集完成后，阅读变更集对应的代码改动；而结对编程是在变更集完成的过程中进行类似的工作。据说，与两位开发人员各自独立工作相比，结对编程会更有效率，且质量更高。要指出的是，测试驱动开发和结对编程还都是存在争议的新技术。

同行评审和结对编程是由人类来完成的。事实上，机器可以在一定程度上代替人。这被称做**静态程序分析**（Static Program Analysis）。相关的工具[1]能够发现一些源代码中的错误和可能的错误并报告，提醒人类注意。

最后再次强调，适时更新工作区。特别是，如果在提交之前，已经做了比较全面的测试，但已经有一段时间没更新工作区了，那么要做一次更新，然后在工作区里编译构建，并作粗略的测试，证明程序可以运行，没啥大碍。这种粗略的测试常被称做**冒烟测试**（Smoke Test）。测试之后再提交。这样就基本可

[1] 比如收费的 Coverity 和开源免费的 Findbugs。

以保证，提交后公共版本库里最新版本是可以工作的，不会给其他开发人员带来很大的困扰。

> **[Subversion]** 注意，这里所说的提交前的更新，是为了质量的缘故。如果提交前，仅仅是因为 Subversion 工作原理而做了一次更新，那就没必要再编译链接和粗略测试了。
>
> **[Git]** 同 Subversion。
>
> **[ClearCase UCM]** 除了更新工作区并粗略测试外，还有一个选择。粗略测试也可以在改动合并到公共流对应的工作区但还没有签入时完成。注意这是在公共流对应的工作区里完成，而非开发流对应的工作区。在确认没有问题后，再真正提交到公共流上，供其他人将来使用。

是的，只是“基本可以保证”。因为这里还有漏洞：万一在这个开发人员检测的过程中，别的开发人员提交了新的修改，而这与正在检测的修改之间有矛盾有冲突呢？如何修补这个漏洞，请看下节分解。

4.3 狭义集成的步骤

即便是大家都挺注意保证提交的质量，但是由于本质上是并行工作，因此还是有漏洞的。团队成员越多，漏洞越多，出现问题的可能性越大。能否考虑某种锁定机制来解决，把并行变成串行？这通常不是个好方法，因为可能会导致开发人员们排队提交……

要么这样吧，专门找个人，时不常地检查一下公共版本库里最新源代码的质量情况。至少是能否编译链接得过，如果过不去呢，就及时修复。如果做得更好些呢，就再粗略测试一下，如果发现有重大问题，就及时改掉。其实这就是集成工程师主要做的事情，也就是常说的（狭义的）集成工作。我们来仔细看看。

第一步，确保开发人员都提交了相关的源代码。作为集成工程师，可以提醒一下大家，本次将集成今天下午 3 点前提交的代码改动。也可能项目经理之

类的领导在确认无误后，告诉集成工程师，没问题了，现在就做集成吧。更为严格的控制是，预先定好本次要集成的工作——这一般是一个列表。在集成开始前，检查一遍，是不是所有该做的变更集都完成并且提交了，并且，列表之外的改动有没有被提交。如果控制得再严一点，那就是，每次提交都必须先获得某种授权，有授权的才可以提交。

大多数情况下都没必要这么麻烦，但有时候确实需要严格的控制。比如，在产品即将上市，或进入维护期后，需要加强这方面的控制。以产品即将上市时为例，这个时候的首要任务是让产品趋于稳定，并且尽快上市，而非进行大幅度的结构调整或增加很多新功能等耗时费力且可能影响产品质量的事。因此，这时候的管理者要站出来，引导开发人员把精力放在当前最要紧的事情上，保证“让产品趋于稳定，并且尽快上市”这样的目标的实现。方法之一就是，在集成这个关口加强控制、加强审批。

> **[Subversion]** 可以在工具中设置一些触发器（Trigger），并编写一些脚本，让工具在执行一些版本控制操作的时候，强制执行某些检查。
>
> **[Git]** 可考虑采用拖入（Pull）的机制：不让开发人员把改动推入（Push）到公共版本库，而是集成人员从开发人员的私用版本库那里，把改动拖入到公共版本库。这样，集成人员只集成通过他检查的改动。另外也可以考虑通过推入权限的管理，或通过 Gerrit。
>
> **[ClearCase UCM]** 除了设置触发器外，还可以锁定公共的地盘（公共流）、仅特定用户可写，可以提交活动，而对其他人只读；或者在创建基线时，对包含的活动有所选择；或者不让开发人员提交到公共流，而是让集成人员在活动被批准时，提交执行活动；或者同时使用 ClearCase 和 ClearQuest。ClearQuest 是 IBM Rational 出品的变更请求管理工具，我们从第 9 章开始会讲到它。

第二步，冻结或者标识将要集成的源代码。这里**冻结**（Freeze）是指临时禁止开发人员继续提交代码修改，而标识通常就是打个标签之类，当然也可以记录下当时的自然版本号。为什么要做这样的操作呢？因为我们得搞清楚，集成的到底是哪些内容。等集成后，我们就可以宣称，我们已经把它们集成好啦！其中，它们是指哪些变更集，集成后形成了哪个整体版本。

冻结或者标识，都是不完美的。若采用冻结源代码的方法，会让开发人员在一段或短或长的时间里，无法提交。而若采用标识的方法，无论是打标签还是记录自然版本号，只要不冻结，那么，当需要为解决集成问题而作出一些代码修改时，有可能这些修改就会混在开发人员为了产品继续向前演进所提交的修改里，而那些修改可能是不希望出现在这次集成里的，或者那些修改中的问题又一次引起集成失败。所以说，这些方法是不完美的，但是，一般来说，这些方法已经足够好了[2]，特别是标识的方法。

第三步，取出要集成的源代码[3]。最好是存放到一个干净的工作区。为的是清清爽爽，别无杂物。这里所说的杂物包括，一些编译的中间文件和结果文件，尚未签入或提交的本地修改，等等。这些可能会干扰接下去的工作，造成错误或混乱。当然，有些时候要做一些妥协，比如，保留编译的中间文件，以便进行增量构建，显著提高编译速度，当然这也有副作用，需要权衡。关于增量构建及其副作用，详见第 5.3 节。

第四步，编译、链接和打安装包。目的是由源代码转换生成可以测试、发布的安装包。这通常统称为**构建**（Build），我们会用专门的章节（第 5 章）来讨论它。

在构建过程中，可能会遇到问题。如果是构建环境、构建过程相关的问题，常常是由集成工程师来解决。而如果是源代码的问题，那通常就要由开发人员来负责解决了。相关开发人员跑到集成工程师这里，指导集成工程师修改一两行代码，或者坐到集成工程师的位子上，把问题解决。当然，更正规一点的方法是，让开发人员在自己的工作区里处理好，然后提交上来。

如果待解决的问题比较复杂或者比较难，预计需要比较长的时间的话，集成工程师可以采用另外一个策略，就是把有问题的提交，甚至是可能有问题的

[2] 在讨论管理交迭的章节里（第 13.4 节），我们会给出其他方法供选择。
[3] 这通常是通过版本控制工具提供的更新或签出操作来完成的。

提交，从本次集成中剔出去。等开发人员弄好了，再重新提交。这样做的理由是，集成是瓶颈，很多人在等着集成的结果，很多人都在意公共版本库里代码的质量，所以集成要尽快完成，哪怕以本次集成少进一个或几个变更集为代价。

> **[Subversion]** 执行一个反向合并操作，就可以把某个已提交的变更集剔出去。
>
> **[Git]** 用反转（revert）命令。
>
> **[ClearCase]** 想把已提交的内容剔出去是比较困难的。在 Base ClearCase 里，需要找到每个相关的文件，分别运行合并命令，跟上若干参数。在 ClearCase UCM 里，有个叫 cset.pl 的脚本能让这个事儿方便一点，但这个脚本并不是“官方”的。

如果在构建这一步遇到了问题，为此动了源代码，那么这之后，回到第一步，从头再来。

第五步，安装并粗略测试。仅仅能生成安装包，通常是不够的。总得能大致运行起来，才像那么回事儿。在这一步，像第四步一样，如果遇到问题，就要解决。如果解决的过程中动了源代码，那就回到第一步，从头再来。

为什么是粗略测试而不是详细测试呢？集成是瓶颈，很多人在等着集成的结果，所以集成要尽快完成，哪怕新版本里带几个小 Bug，也就忍了吧。更详细的测试，可以等新版本出来之后再慢慢测，Bug 可以随后再改。详见下一节。

第六步，标识和储存集成成果。集成成果至少有两个，一个是源代码的整体版本，一个是生成的安装包。它们都要被合理的标识和存储，供大家使用。源代码打了合适的标签或者记录下自然版本号就好啦。注意通过特定的标签命名约定或标签属性等方法，把这时候打的标签与前面第二步打的标签相区别，让大家一眼就能看出来哪些标签通过了集成，哪些（还）没有。

至于安装包，通常放到某个共享目录下存储。这个话题，在第 5.5 节还会详细讨论。

第七步，通知相关人员，本次集成完成。这可能包括开发人员、测试人员

和各级各类领导同志。通知的方式可以是电子邮件，也可以是 RSS 订阅之类。除了通告集成完成之外，还应该告知集成成果的名称和存储的位置。另外，还可以在此简述本次集成对应完成了哪些开发工作、修复了哪些 Bug，等等。

总之，狭义的集成，其基本的目的是产生包含了最近提交的改动的，一个有一定质量的整体版本。这个版本我们通常叫它**基线**（Baseline）。严格地说，基线既包括了对应的源代码整体版本，也包括了编译构建生成的安装包。

4.4 在基线产生之后

为什么需要集成工程师集成？为什么要产生基线？原因之一是，开发人员要用。我们以前讲过，开发人员从公共版本库中拿到最新版本，用来更新自己的工作区。其实这个说法并不严格。有些时候，开发人员是想拿到别人刚刚提交的修改，也就是公共版本库里最新的内容；而有些时候，开发人员是想拿到最新的基线，这样拿到的代码比较靠谱，没什么大问题。

事实上，即便是开发人员总是去拿公共版本库里最新的内容，集成工程师的工作对他们也是有意义的。因为做狭义集成能够发现并修复一些已提交代码中存在的严重问题，不再是每次都等到某个（或者某几个）不幸的开发人员撞见了。在统计上，即便开发人员不直接使用基线，他还是从狭义集成中受益。

另一方面，测试人员也需要使用基线。（狭义的）集成产生了具有一定质量的基线。测试人员对其中的一些基线进行全面而深入的测试。在测试过程中，不断发现问题，也就是常说的 Bug。测试人员在缺陷跟踪系统里（详见第 9 章）记录下来这些 Bug，通知开发人员整改。开发人员修复若干 Bug 后，集成工程师集成，产生新的基线，再次送交测试人员测试。

如此往复，直到有一天，某个领导查看了测试报告等资料后宣布，质量过关啦，拿出去卖钱吧！于是产品就对外发布啦（见图 4-2）。

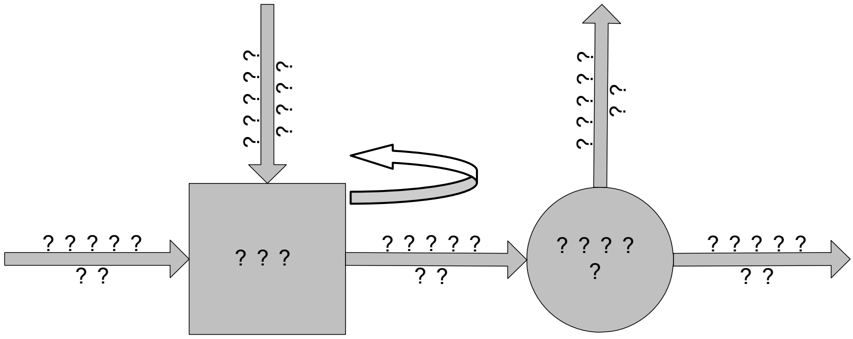


图 4-2　开发-测试-发布[4]

4.5　质量保证：集成前、集成中、集成后

发现没有，广义的集成其实发生在三个阶段中。第一个阶段，狭义集成之前。其实就是开发人员的改动提交到公共版本库之前。在这个阶段，开发人员更新工作区、代码评审、编译链接、做一系列的自测试，等等。这些都是为了发现和改正代码中的问题，包括集成问题。

第二个阶段，狭义集成过程中。集成工程师收集最近提交的代码改动，在一起编译链接，并粗略测试，然后发布基线给大家。这些工作的主要目的也是发现和改正代码中的问题，其中主要是集成问题。

第三个阶段，狭义集成后。测试人员使用基线进行系统全面的测试，发现问题，并交由开发人员改正。

这三个阶段，都在做质量保证工作，都在消灭集成过程中的问题，直到最后，达到一定的质量标准，集成完成。注意三个阶段的质量保证工作，有它们各自的特点：

[4] 注意，这幅图是一个简化的图。比如，安装包的存储、测试、发布，通常是先后进行的，图中没有表达这一点。

- 第一阶段，狭义集成前

在第一阶段里，不论是代码评审还是自测试，质量保证工作主要围绕着发生的代码改动及其周围进行，进行深入的检查，并改正问题。

这样的工作可以有效率地发现问题，因为问题通常出现在代码改动及其周围，一抓一个准儿。

这样的工作也可以有效率地解决问题，不必把集成工程师或测试人员折腾进来，也不必在缺陷跟踪工具里走一系列流程。集成工程师或测试人员来掺乎的话，他（们）得去判断该去找谁，再找到他，再沟通交流，等完成了，再验收，有问题，再去找，等等。如果开发人员在变更集提交之前就弄好，那这些事儿就都没了。

如果一些重要问题没有在此发现，留到第二阶段狭义集成的时候暴露出来，会增加集成工程师的负担，而他的工作是串行地解决出现的问题，一个问题一个问题地分析并解决。不像各个开发人员在提交前是并行地发现和解决问题：每个开发人员随时都在解决，通常互相之间不用等待、排队。因此，集成工程师的工作，容易成为瓶颈。即便没有形成瓶颈，他耗费的时间，通常比某个开发人员耗费的时间更能影响项目的进展，因为大家都在等着集成工程师的工作成果，也就是基线。因此，要尽量避免重要问题到狭义集成的时候才暴露出来。

而如果一些在此阶段发现和改正的成本并不高的问题留到了第三阶段才去发现和改正的话，除了发现和改正的成本可能升高外，还有可能影响到了开发人员的开发，或影响到了测试人员的测试。开发人员会被不相干的问题所打扰。比如，想给图片显示软件加个随机播放功能，可是即便是顺序播放时，软件都会不稳定，因此开发随机播放功能遇到问题时，开发人员就会困惑：是我的代码有问题还是别人的代码早就有问题？测试人员也可能被影响。假定某个游戏

有 5 个场景，顺序出现。如果第 2 个场景有个 Bug，程序总是到这儿就退出了，那么剩下的 3 个场景就没法测了，只能等这个 Bug 修复了才能继续测。

- 第二阶段，狭义集成中

在第二阶段里，质量保证工作主要是在有限的时间里，保证程序在整体上没什么大问题。

加上这一道检查，能够防止更多的人被同样的重大问题影响。这些重大问题，比如编译不通过，会对开发人员有广泛的、严重的影响，应该尽量避免。

还要注意，此阶段的检查不宜用太长的时间。太长的时间，就意味着等待新的基线需要很长的时间，延误项目。相关的，此阶段的质量标准，也不宜设得太高。太高的质量标准，基线产生过程中就会遇到太多的问题必须要解决，这使得基线的产生更为困难，需要更多的时间。

- 第三阶段，狭义集成后

在第三阶段里，质量保证工作是对程序进行全面的、广泛的、细致的检查，找出问题，改正问题。

为什么不在第一阶段进行“全面的、广泛的、细致的检查”？因为没效率。在第三阶段的某次“全面的、广泛的、细致的检查”发现了来自各个开发人员提交的、积累下来的 20 个 Bug，但如果每个开发人员都在第一阶段进行一次“全面的、广泛的、细致的检查”，那可能只能比“围绕着发生的代码改动及其周围”进行的检查多发现一两个 Bug。而一次“全面的、广泛的、细致的检查”可能成本很高，不是用人时来衡量，而是要用人月来衡量。

如果硬要开发人员这样没效率地工作，开发人员就会有一种自然的倾向，攒上很多代码改动，一起测试，一股脑提交。于是，开发人员之间因为任务相互依赖而互相等待的情况就增加了，项目关键路径延长了。并且，开发人员分

头开发，前进很久才合并，会导致更多的合并问题……因为这些违背了持续集成（详见第 6 章）这一基本原则。

总之，各个阶段的质量保证工作既不能缺，也不能走过头，要恰到好处。

集成之道，中庸之道。

5

第 5 章　从源代码到运行中的程序

在上一章里，我们学习了关于集成的一些基础内容。在下一章里，我们将学习持续集成。那么，这两章之间，为什么是貌似和集成关系不大的本章呢？这不是加塞儿吗？

5.1 管理生成转换过程和相关资产

前几章讲的内容，似乎都是围绕源代码的管理，比如源代码的版本控制，源代码的整体质量啊之类。没错，围绕源代码的管理的确是软件配置管理的重头戏。然而，其他形式的软件资产也同样需要管理。典型的，编译构建生成的安装包就需要管理。此外，软件的从源代码到运行中的程序这一生成转换过程本身也需要管理。这些都是本章的内容。

开发人员阅读和编写程序的源代码，但源代码通常不是可以直接运行的程序。源代码需要经过一系列的生成和转换过程，才能真正跑起来。下面是一个典型的生成转换过程：

- 第一步，每个源代码文件分别编译（Compile）成目标文件。
- 第二步，目标文件被链接（Link）到一起，形成可执行文件及其附属文件。
- 第三步，生成安装包，以便在任意一台合适的机器上方便地安装。
- 第四步，在合适的环境中安装这个安装包。
- 第五步，在合适的环境中运行该程序。

注意，第一步和第二步具体是什么样子，是与编程语言密切相关的。C/C++语言中，使用编译、链接这样的术语，换一种语言，就可能换一种说法了。甚至，不少脚本语言不需要这两步，而是在运行脚本的时候解析。第三步有的时候可以省略。特别是在程序开发的过程中，开发人员自测试时有可能不是这么麻烦，只是在第四步时简单地把可执行程序复制到某个目录下。甚至连复制都省略了，在原地直接运行第二步中生成的可执行程序。

前三步，统称为**构建**（Build）。构建首先要保证可重复性。也就是说，相同的源代码，不论由谁构建，在哪台机器上构建，在这之前构建过什么，都应该

得到相同的构建结果。具体内容请见本章第 5.2 节。构建过程需要尽量快，因为等着用构建结果。如何做到呢？增量构建是个经常考虑的方法，具体内容见本章第 5.3 节。还有一些其他方法，详见本章第 5.4 节。构建的结果，通常就是安装包，需要版本管理：把它放在哪儿？版本名称怎么定？如何记录或体现源代码的版本？相关内容见本章第 5.5 节。

第四步和第五步通常被称做**部署**（Deploy）。当测试软件开发成果时，测试人员把程序部署到测试环境。开发人员自测试也同样需要测试环境。而当用户买到软件后，用户会在一定的环境中安装并运行它。关于部署的管理，重点是对运行环境的管理，以保证程序安装到合适的环境中，并在合适的环境中运行。详见本章第 5.6 节。与部署相关的更多内容，将在第 18 章介绍。

5.2　保证构建的可重复性

构建的输入，是产品的全部源文件，可能还有文档、数据等。构建的输出，通常是安装包。如果构建的输入是一定的，那构建的输出就应该是一定的，就是我们想要的那一个。不然，构建就变成了不可预知的，捉摸不定的。同样的源代码，也许这次构建产品就有某个 Bug，下次构建这个 Bug 又没有了。再构建一次，说不定又冒出来了……这样的构建，会让人很难接受。

不仅集成工程师这边构建的可重复性是个问题。开发人员这边的构建，应该跟集成工程师构建的结果一样。不然，他们之间就可能会打架：一个说能编出来，一个说编不出来；一个编完了能运行，一个编完了运行不起来。

时光的流逝可能会让这个问题更严重。源程序的某个版本，现在能构建，能重复构建得到相同的结果，那半年后呢？一年后呢？更长的时间后呢？可能公司统一使用的编译器版本升级了。谁也记不得当年使用的编译器的版本了。更要命的是，那个古老的程序中一个重要的用户偏偏提出要修复一个 Bug……

对从源代码到安装包这一生成转换过程，要有足够的管理，确保其可重复性。那么有哪些工作要做呢？为了让结果是明确固定的，我们需要：

第一，原材料是固定和明确的。要保证产品的全部源代码的版本是固定和明确的。如果有其他输入，比如数据、文档等，也应该是固定和明确的。如果连输入都不确定，何谈输出？

第二，工具是固定和明确的。比如，用的是什么牌子的、哪个版本的编译器？用的是什么牌子的、哪个版本的打包工具？甚至，更广泛的，用的是什么牌子的、哪个版本的操作系统？如果这些不同，都有可能导致相同的输入产生不同的结果。这是软件方面；而在硬件方面，有时也需要做出一定的说明。比如，是 32 位机还是 64 位机？最低的硬件配置是怎样的？

有些原材料，是工具自带的，暗含的。或者在某种意义上说是公共的。比如，编译工具自带的静态链接库；或者是被广泛使用的，比如，由另外某个厂家提供的静态链接库。这些，也都有牌子、名字的问题，以及版本的问题。如果它们变化，结果就会变化。如果它们不确定，结果就会不确定。

第三，生成方法是固定和明确的。方法步骤、参数设置都应该是固定和明确的。比如先使用哪个工具，执行哪个操作，再使用哪个工具，执行哪个操作。这可以在具体执行编译、链接和打包的命令及其参数上体现出来，也可能在 Makefile 文件、Ant 文件、相关脚本上体现出来。这也和对工具和环境进行配置时，设置的参数有关，其中一个例子是操作系统的环境变量。

总之，根据固定和明确的输入，使用固定和明确的工具，按照固定和明确的方法，产生的输出，就是固定和明确的了。这是我们想要的。

想要做到这一点，一个基本的方法是文档化。尽可能地文档化构建过程，详细说明如何构建，直到任何人在任何时间，都可以根据这个文档，恢复当年的构建环境，并正确执行当年的构建过程，产生与当年一样的结果。

一个更好的办法是自动化，尽可能地自动化。因为对于重复性的工作，人有时可能会犯错误，但是由工具自动地完成，犯错误的概率就大大减小了。把在过程中需要人工输入几十行命令的构建过程，变成只需要在屏幕上按个按钮就能自动运行的构建过程，过程本身执行的正确性一定会大大提高。把需要人工设置的构建环境，变成由一个脚本自动设置，那么选错工具或选错工具版本之类的几率就会大大降低。

把构建用到的工具、用到的公共原材料，放在公共的地方，也是个不错的技巧。不论是谁来做构建，集成工程师或开发人员，都必须到那个公共路径下取得工具，取得公共静态链接库，等等。这可以强制保证，每个人都在使用相同的工具和环境。不过，有些时候，这会有技术上或其他方面的困难，比如与服务器连接，远程运行的速度可能比较慢。

更进一步，可以考虑把尽可能多的东西，包括原材料、工具、配置参数、Makefile、执行脚本、构建过程文档等，放到版本库里。使得一旦导出了特定版本的源代码，也就导出了对应于该版本的这些内容。这些内容可能会随着版本的变化而变化，而现在把它们和源代码的版本绑到一起了，就保证了构建的可重复性。用这个方法也要小心过犹不及。比如，把操作系统的安装包放进去，好像就没多大意义，就不如在构建说明文件里简单地写上，“本次是在 MS Windows XP SP2 上编译的”[1]。

5.3　全量构建和增量构建

构建分为**全量构建**（Full Build）和**增量构建**（Incremental Build）两种基本类型。这主要指编译链接的过程。

全量构建是从头来过，从每一个源文件的编译开始。也就是说，是完全重

[1] 还有其他方法，比如，在 Makefile 中自动进行检测，若发现构建工具、操作系统等的版本与要求的版本不一致时，停止编译并报错。

新编译源代码，进而链接、打包。而不是借助于以往构建中留下的已有的或许可以重复使用的中间成果。

而增量构建则不同，它尽可能地利用上次构建的成果。增量构建的输入不仅包括原材料，还包括上次构建中产生的中间产品，比如编译产生的很多中间文件。增量构建中，会仔细考察每一件要做的事情，比如根据某个源文件生成一个相应的中间文件。仔细考察，如果这回再一次做这件事情，再次根据某个源文件生成一个相应的中间文件，所生成的中间文件，会不会与上次所生成的完全一样。如果预计完全一样，那么何必再一次做这件事呢？把上次的直接拿来用就是了！这可是个省时间的方法。

对于某个源文件，如何判断如果再编译生成一次，其结果能否与上次的结果相同？基本的判断原理是，如果某个源文件及其所包含的头文件，在上次构建之后，其内容没有改变，那么这个源文件的生成文件，也就不会改变[2]。而如何判断某个文件，在上次构建之后，其内容有没有改变呢？一般来说，如果这个文件的修改时间，在上次构建之前，我们就认为它的内容没有改变[3]。

是的，增量构建与全量构建相比，优势在于快。而它的劣势是，不是那么可靠。这里面有很复杂的原因，开发人员一般都知道一些，上文的脚注中也提到一些。总的来说，难点在于，当一个源文件变化的时候，要找出所有要重新编译生成的文件——这被称做影响性分析。

> **[Clearmake]** Clearmake 是 ClearCase 自带的。使用 Clearmake 而非 make，有助于提高增量构建的可靠性。Clearmake 根据源文件的版本信息，而非根据源文件的修改时间，来判断该文件是否应该重新编译。在考虑使用它的时候，你一定要实测一下它在你特定的应用场景中的工作效率。

[2] 注意这个大致的叙述并不严格。比如，如果包含的静态连接库变化了，那么也需要重新编译。资源文件的变化亦有可能导致需要重新编译。而如果编译器、编译参数等变化了，生成文件也有可能变化。

[3] 这也是不严格的。如果开发人员从别的地方复制过来一个旧版本，或者当他在版本控制环境下，在工作区里，更新到历史上的一个基线上的时候，那么，通过查看文件修改时间的方法，就有可能得出错误的结论了。

通常，我们强调构建速度的时候，会倾向于使用增量构建，比如开发人员给自己构建的时候；而强调构建的正确性的时候，会倾向于使用全量构建，比如构建发给客户的版本的时候。

通常，系统集成时集成工程师所作的构建是全量构建，因为要保证构建的正确性。特别是，当构建的结果送去系统测试，将来可能会发给客户的时候。但有些时候，可能也需要通融。比如，对于规模比较大的产品，如果集成发生频率很高，特别是采用持续集成（参见第 6 章）的方法，那么每次集成，都重新获取全部源代码，全部重新编译链接，可能代价会比较大，或者难以做到。这时，可以有选择地只对重要的构建，比如用于发布以及重要的系统测试的构建使用全量构建，确保构建的质量和对应基线的质量；而对于其他大多数构建，用增量构建，以节约时间和资源，哪怕它不是绝对可靠。

5.4　让构建更快

如果目前你参与研发的产品，其构建所需时间在几分钟之内，那么你不需要特别关注本节内容。而对于参与大型项目开发的同志们，这一节则特别重要。大型软件产品的构建，可能需要数个小时，甚至数十个小时。如何尽可能地优化和提高速度，就成为了重要的课题。

我们已经了解到，自动化是保证构建的可重复性的一个重要途径。事实上，自动化也是提高构建的速度和效率的重要途径。不仅仅是使用 Makefile 和 Ant 这样的工具，来运行自动化编译过程、自动化链接过程。最好是把构建过程从头到尾都自动化，只在最开始发个启动信号，然后就等着收结果了。这肯定要比手工敲入命令，等待某个中间步骤的结果，对结果进行分析判断，继续敲入下一行命令……这种方式快。而更进一步，可以让构建工作自动定时启动，自动报告结果。我们在讲持续集成的时候会详细讲（见第 6.3 节）。

除了自动化之外，还有哪些方法能够提高构建速度和效率？提高硬件性能是很自然的途径。更高主频的 CPU、双核或多核、更大容量的内存和更快存取速度的硬盘，这些都可以提高构建服务器的性能。

另一方面，如果构建过程中需要与其他机器发生交互，那么提高网络速度将加快访问构建工具和原材料的速度，因而使构建更快。减少对网络的依赖，用本地复制代替，也会使构建更快，因为本地访问总是比网络访问快。

而提高专一性是另一个容易想到的途径。不要让构建服务器干别的事情。同时，尽量减少在同一台服务器上同时运行的构建任务的数量。

更进一步的想法是，把构建任务分解，并行完成。可能是使用具有多个 CPU 的服务器，把构建任务分解，分配到多个不同的线程或多个进程上，这些子任务并行完成。还有可能，是把构建任务分解到不同的服务器上运行，这些服务器通过网络或其他连接方式，相互配合，并行工作，共同完成一次构建。这种方法，与集中式的方法（多 CPU）相比，在效果相同的情况下，硬件总体造价可能会低很多。若干台运行 Linux 操作系统的普通个人计算机，比起有几十个 CPU 的中央服务器，可便宜多了。但是，要实现分布式的构建，其软件实现难度则大了很多，可能需要一些高端软件的支持。

有没有可能减少重复工作？有没有可能不再重复去做过去已经做过的工作，把当时的工作成果直接拿过来？这正是增量构建的思路，我们已经讲过。需要注意的是，增量构建通常是指，重复使用自己过去所做的构建中的成果；而这些成果，是在自己所在的目录中，在自己的工作区里的。

那么，更进一步，有没有可能，重复使用别人构建的中间成果，共享它们？这些中间成果，可能是某个.c 文件编译生成的一个.o 文件，也可能是某个源代码模块编译生成的一个静态链接库.lib 文件。这些中间成果，可能在同一个服务器的不同目录下，可能在不同的服务器上，也有可能被特定的工具所收集和管理

了，这样做就是为了重复使用它们。这是一个伟大的想法。一些工具已经可以提供这样的支持。

> [ElectricAccelerator] Electric Cloud 公司最耀眼的产品。通过一系列加速技术，它能够大幅度提高构建的效率，比如提高 20 倍。当然，它的价格也不菲。
>
> [Clearmake] Clearmake 支持 Unix 平台上的并行分布构建：同时利用多个 CPU，或者利用多台计算机。Clearmake 可以重复使用别人构建的中间成果。这是因为，Clearmake 记录下了每个派生对象（Derived Object，DO），是由哪些源文件生成的，而派生对象和源文件一样，也存储在 ClearCase 版本库里。不过，再次提醒，在决定使用这些功能之前，你最好先实测一下它的运行效率，看它是否真的能降低构建的时间。

增量构建、分布式构建……我们在考虑这些方法的优点的时候，不要忘记它们可能带来的麻烦。由于方案的复杂性，构建可能变得脆弱和不稳定。由于方案的复杂性，重复某个久远历史上的构建可能会变得更困难。

上面讲的都是如何让构建更快，事情还有另一方面，即让排除构建错误更快。本次构建失败，到底是构建系统不稳定出了问题，还是源代码写得不对？到底是哪个模块，哪个源文件带来的问题？为此，我们需要构建工具的使用者们熟悉构建。而从构建工具的角度，关键的一点是，提供信息，方便使用者定位错误。这就涉及到构建过程的记录，特别是构建错误的记录。比如，构建什么时候开始的，什么时候结束的，在哪台机器上执行的，相关构建工具的版本是多少，具体编译了哪些文件，先后次序是什么，有多少个错误和警告，各是什么类型的，出现在哪个地方，什么环节，哪个模块，等等。相关细节在此不展开。

5.5　安装包的版本控制

源代码是软件资产，需要版本控制。源代码经过构建，转换为安装包。安装包也是软件资产，也需要版本控制。保存安装包的主要好处是，将来在需要

它的时候，可以迅速准确地得到这个安装包。如果在将来需要它的时候，现找历史上的源代码，现从源代码开始编译、打包，那么会耗费时间。对于企业来讲，时间就是金钱。

开发人员做构建，产生安装包。集成工程师也做构建，产生安装包。对这两者的管理力度是不一样的。原因是，开发人员产生的安装包，通常只是供他自己一个人用。而集成工程师产生的安装包，反映了开发的某个基线，众多开发人员可能用它作为参考，测试人员也可能用它来测试。显然，集成工程师产生的安装包更为重要，需要放到更容易被大家拿到的地方，更安全不容易被破坏的地方，并保存相对来说更长的时间。

安装包如何保存？放进版本库里通常不是明智之举。版本库通过增量存储来减少存储所耗费的磁盘空间，但是这通常只对于文本文件有效。对于文本文件，比如源代码文件，只记录文件的哪几行更改了，而非整个新版本，以此来减少存储耗费的磁盘空间。对于二进制的安装包，这个方法则难以实现。而另一方面，对于安装包，很多历史版本，比如那些没有发布给用户甚至没有用于系统测试的安装包，是不值得长期保留的，应该适时删除或移至其他存储介质，以免占用大量的磁盘空间。版本库原则上保存全部历史版本，如果想删除已经没用了的历史版本，会很麻烦。不像在平常的文件系统上。因此，用版本库来存储安装包，反而带来不便。

在像 Git 这样的分布式版本控制工具里，存储安装包通常就更不明智了。分布式版本控制工具在每个开发人员的机器里都会存储一个本地版本库，其内容与服务器端的公共版本库基本相同。可以想见，每当提交一个安装包到服务器端，不久以后，所有机器上的存储库都会因此而增大尺寸。其间，网络流量也不小。如果安装包比较大，提交得又比较频繁，那将会有严重的问题。

那么，安装包该如何保存？作为一种简单的方法，安装包可以保存在共享目录下，该共享目录可在局域网上共享。要合理地设置访问权限，包括读权限

和写权限。更精细的设置，还可能包括对删除、重命名、移动等权限的设置。应该只容许确有必要的少数人可以进行“危险性”的操作。除权限设置外，还要考虑适当的备份，防止因操作系统或硬件故障，或其他天灾人祸，共享目录下的内容丢失。这方面，通常可以咨询 IT 部门或服务器管理组的同事，他们对这些比较熟。

> **[Subversion]** 如果你使用 Subversion，那么一般来说，不应该把安装包放进 Subversion 里，而应该放进共享目录里。
>
> **[Git]** 请尽可能不要把二进制文件放入 Git 中，除非它很小或者绝少更新。
>
> **[ClearCase]** 如上一节提到的，ClearCase 不仅存储和管理源代码，也存储和管理由源代码生成的内容，这些内容被称为派生对象，包括可执行文件、库文件、数据文件等。ClearCase 并不特别地支持安装包的管理，但你可以认为安装包也是导出对象的一种。

还可以做得更好。有一些专注于安装包管理的工具：生成安装包、存储安装包、部署安装包、并记录相关信息。这样比较方便，也不容易出错。在第 18.1 节，我们将详细介绍。

当讨论安装包的版本控制时，还有一个重要的话题，就是记录安装包版本和源代码版本的对应关系。人们需要知道这种对应关系，顺藤摸瓜从源代码整体版本找到对应的安装包，或者反方向，从安装包找到对应的源代码整体版本。

比如，测试人员在安装包的某个版本中发现了一个 Bug，这个 Bug 在上一个版本中还没有出现。于是，开发人员就得到了一个关于这个 Bug 的线索：一定是这两个版本之间的代码改动，导致了这个 Bug 出现。为此，他们可能会查阅对应的两个源代码基线之间的变更集，发现更多关于这个 Bug 的线索。

那么，如何记录安装包版本与源代码版本的对应关系呢？在安装包里添上说明文件，说明源代码版本？在安装包之外的某个地方记录每个安装包版本与每个源代码版本的对应关系，形成表格？都值得考虑。然而，应该首先考虑最简单的方法：源代码版本与安装包版本，两者使用相同的版本名称。

5.6 管理测试运行环境

运行环境是指产品运行的环境。对于最终用户来讲，这个是唯一关心的环境。但是不仅仅是最终用户关心它；测试人员也关心它，因为需要在这之上做全面的测试；开发人员也关心它，因为需要在这之上调试和测试；集成工程师当然也关心它。

运行环境大致包括：产品运行的硬件环境，比如，最少需要什么样的硬件配置，以及推荐什么样的硬件配置；产品运行的软件环境，比如，操作系统的类型，以及最低版本；是否需要预先安装一些平台性的软件，比如 Java 的运行平台，Perl、Python 等语言环境，以及 Oracle 等数据库；是否需要预先安装一些动态链接库；需要做哪些相关的配置和设置。总之，是产品运行所需要的所有前提条件。

运行环境应该明确地让用户知道，否则滋生抱怨。对于其中特别重要的方面，应该在产品宣传资料上、销售合同上、外包装上就给出。对于细节，可以考虑在发布说明文件中给出。

让用户明确地知道所需运行环境，前提是，在产品研发过程中，明确地定义所需运行环境。并且，在分析、设计和编码中，考虑到所需运行环境。在开发人员自测试和测试人员系统测试时，在所需运行环境上测试。

如果运行环境中的一些选项可以有不只一个取值，那么要考虑在不同取值的情况下测试。比如打算在 MS Windows 2000、MS Windows XP、MS Windows 7 这几个相似的操作系统上运行，那么，要考虑在几个操作系统上都做测试。当然，这并不一定是穷举。比如，如果该程序需要与某个数据库配合工作，那么并不意味着需要在该数据库的所有发行版本上，比如 1.0.0、1.0.1、1.0.2、1.0.3，

等等，都要做测试。那样的代价太高了。而边际收益递减[4]。

当我们谈到测试时的运行环境时，还要讨论测试使用的数据和配置。因为使用不同的数据和配置，可能产生不同的测试结果。测试使用的数据和配置，也许是文件，也许在数据库里。不论是哪种形式，都要保证它是明确的、可复现的、标准的。如果做得更好，那么这些数据和配置应该能够方便地部署到测试环境中，比如用一行命令就能完成。

最后，让我们回到开发人员的视角。开发人员工作在一定的环境中，他需要文本编辑器、版本控制工具、同行评审工具等一系列工具，支持他与团队成员合作，修改源代码；他需要构建工具等构成的构建环境，帮助他构建，把源代码转化为可运行的程序；他需要合适的测试和调试环境，以便研究程序运行时展现的功能和性能。对这些环境的管理，颇有一些共同之处：做记录、标准化、可重复、自动化等等。

通过本章的讨论，我们对从源代码到运行中的程序这一生成转换过程有了大致的了解。我们已经做好准备，迈向持续集成，也就是下一章的内容。

[4] 边际收益递减是一个经济学名词。如果你还不太了解，那就到网上查查。也许它会改变你思考配置管理问题的方式。

6

第 6 章　迈向持续集成

持续集成可是个热门词汇。让我们看看它到底是怎么回事，为什么要实现它，如何才能实现它。

6.1　及早和经常地集成，持续集成

过去，人们试图用瀑布模型来开发软件。把项目分成若干阶段，每个阶段有明确的任务、明确的输出：确定需求、总体设计、详细设计、代码实现、系统集成、系统测试、发布。这种方法已经被证明是低效的。今天的世界已与那时有了很大的不同。别的先不说，在本章里，我们来看一下集成这方面，如今的策略是什么样的。

及早和经常地集成，这是如今被广泛认同的最佳实践。我们已经讨论过的，在集成中，会发现接口中存在的问题、对设计的误解、变更集间配合的问题，等等。这些问题早发现、早解决，要比拖到最后，再去理清一团乱麻要好。

事实上，及早和经常地集成，还会显著减少出现这些问题的总量。这可以从数学上进行证明，但这里我们试着直观地理解一下：如果开发人员长期隔离自己，那么他就会越来越不知道前进的大方向，每迈出的一步，也就越有可能走错。同样，如果长期不集成，那么各个开发人员的每一点新的成果，也就从概率上更不容易和其他开发人员的工作成果相融合，更容易出问题。

及早和经常地集成，包括三个方面的含义：其一，开发人员要经常提交，完成一个小任务就提交，尽量不要攒一大堆再提交。攒一大堆，就会有很长的时间，别人见不到你所做的工作，与你的工作之间，有了越来越多的不协调和分歧。

要注意的是，经常提交，并不意味着，在一个变更集都还没有完成的时候就提交，去和整体集成。总要完成一小块工作单元，产生对别人有价值的东西再提交。总要让它具有了一定质量，不会造成对别人困扰的时候再提交。这里的“别人”主要指的是开发人员和测试人员。但如果产品都快上市了，那就还要多考虑最终用户，不要留下半拉子工程给用户，给用户造成困扰。

其二，如果因为某种原因，确实需要开发一段时间再提交，那么在开发过程中，要及时做更新，不要留到提交前，或者干脆不更新直接就提交。讲解第一点时提到的“越来越多的不协调和分歧”，请你尽量自己消化，不要留给别人。而自己消化的时候，不要攒一大堆再消化，适当地细水长流要好一些。

其三，在集成工程师这一端，要经常（狭义的）集成开发人员提交的内容。事实上，第三点是人们通常更关注的内容。

及早和经常地集成，典型的相关词汇是**每日构建**（Daily Build）或称每夜构建 （Nightly Build）。注意这里说的构建，通常也包括了粗略测试。每天都做一遍集成，这够了么？在**持续集成**（Continuous Integration，CI）看来，还不够。持续集成认为，集成可以更为频繁，也应该更为频繁。

持续集成的频率，通常比每天更密集。平均每半天、每两小时、每一小时，甚至更短。而如果每当公共版本库接收到开发人员的提交后，就进行集成，那是最好啦。这样做就是为了尽可能早地发现版本库里最新源代码上的问题，尽可能快地纠正。这样，在绝大部分时间里，版本库里的最新源代码都没什么大问题。

前面我们讲过，开发人员可以选择用公共版本库里最新的源代码更新自己的工作区，也可以选择用公共版本库里最新的基线更新自己的工作区。前者取得最新代码，却缺乏质量保证。后者有质量保证，却不是最新的代码。两者各有优缺点，没有完美的方法。而如果集成的频率很高，这个两难问题就解决了：开发人员在用最新的源代码更新自己的工作区时，比较放心，十有八九，是没问题的。而如果用最近的基线更新自己的工作区时，又可以说，嗯，这个已经很新了！

持续集成还有其他的好处，比如，项目管理者能够在项目开发过程中就看到货真价实的研发成果，看到功能不断丰富，看到质量不断提高。他心里踏实，

项目风险可控。最终用户或者最终用户的代表也可以在项目早期就看到部分成果，有真切的感受，及早了解到这是不是他想要的，早点说改或许还来得及。

6.2　化繁为简

越大越复杂的软件开发项目，越能从持续集成中获益。然而遗憾的是，越大越复杂的项目，实现持续集成也就越难。从本节开始，我们来讨论如何实现持续集成。而其难易程度，取决于你所在的项目本身。

比如，如果是个小项目，通常它的源代码就在一个版本库里。而大型项目的源代码因为种种原因（见第 15 章和第 17 章）有可能分布在多个版本库里，甚至分布在不同牌子的源代码版本控制工具的版本库里。这使得不论是开发人员还是集成工程师，在集成过程中，进行版本控制操作的时候都比较复杂烦琐，容易出错，并且不容易自动化。

如何应对？在可能的情况下，在一个项目中，尽量只使用一个版本库。如果做不到，也尽量只使用一个牌子的源代码版本控制工具的版本库。实在不行，再用多个牌子的工具。如果确有必要，不得已用多个版本库甚至多个牌子的工具，也尽量在上面进行一些封装：用一条命令、一个操作，代替若干条命令、若干个操作，让使用者用起来比较简便。从集成工程师的角度来看，最重要的是，能够方便地从版本库中取到全部源代码，能够方便地标识作为整体版本的基线。

如果是小项目，它可能只使用一条主干（主干、分支的概念详见第 7 章）。而大型项目，可能因为各种不同的理由，使用很多分支。分支确实有用，但分支是有成本的。用这么多的分支，真的是有必要的么？尽可能，只使用一条主干。只在确实有必要的时候，谨慎地引入新的分支。另外，可以考虑用一些脚本封装，来向开发人员等使用者屏蔽一些由分支带来的复杂性。

如果是小项目，通常编译构建就是点个按钮或敲一行命令的事儿，而不是繁杂的步骤，因而不容易弄错。构建命令的参数、可选项什么的没有或者很少，不容易弄错。并且编译构建时所依赖的第三方库之类的也很少或没有，不容易弄错。而大项目，可能刚好相反。同样的源代码，在不同的机器上，由不同的人来做，可能有的人编得过，有的人编不过；有的人编好了能运行起来，有的人却运行不起来……

在第 5.2 节里，我们介绍了如何保证构建的可重复性，以保证如果构建的输入是一定的，那构建的输出就应该是一定的，就是我们想要的那一个。总的来说，要保证构建环境是标准的、构建输入是标准的、构建过程是标准的，等等。可考虑用脚本来封装，实现这样的标准，让开发人员、集成人员能够轻易地搭建起标准构建环境；让开发人员、集成人员能够轻易地执行标准构建。

不光是构建，测试也是如此。测试环境，不论是基于真实硬件的环境还是模拟器中的环境，应该是标准的；测试数据应该是标准的；测试用例应该是标准的。这样，测试就是可重复的。应该有简单又简便的方法，建立标准测试环境，导入测试数据。应该有简单又简便的方法，把待测程序安装部署到测试环境中。应该有简单又简便的方法，自动顺序测试各个测试用例……

总之，不论是源代码版本控制，还是编译构建管理，还是测试运行时的管理，这里强调的都是，尽可能用简洁的方法，不要引入不必要的复杂性。因为复杂往往意味着容易出错、意味着不稳定、意味着难以重复、意味着高昂的人力成本。然而，复杂性不是总能避免的。特定项目的特定背景，决定了其必要的复杂性。于是，我们就要使出第二招：尽可能封装，把复杂性隐藏起来。使得在“外面”看起来，工具、方法和过程是简单方便的。这样，我们才愿意有更频繁的集成，而不至于因为频繁的集成而在烦琐、枯燥、易错的重复性工作中耗费太多的心力。

6.3　全程自动化

沿着上一节的思路走下去。不论是广义的集成还是狭义的集成，在集成过程中，都是有很多重复性的工作的。当集成的频率提高后，这些重复性的工作就变得更多。如果这些重复性的工作由人来完成，那么就会显著提高项目的人力资源成本，甚至延长项目时间。这是我们不愿意看到的。因此，当我们向持续集成迈进的时候，自动化就变得特别重要。

其实上一节就是在谈自动化。一行命令，实现自动化编译；一行命令，实现自动化链接；一行命令，自动生成安装包；一行命令，准备好测试环境；一行命令，部署程序；一行命令，进行自动化测试；等等。然而，这些是不够的。

它们之间不是自动化的，需要有一个人盯着。集成工程师敲一行编译命令，然后喝茶看报纸；等编译结束了，再敲一行链接命令，然后挖地雷；在集成工作结束前，他还吃了些小点心，网聊了一会儿，起来散了散步，发现相邻部门新来的女同事很迷人。但是，另一方面，他不得不工作到晚上 12 点，因为大部分时间，都在等待中度过了。

如果集成的各个步骤之间，也能够自动化，那就完全不同：只需要做好设置，后面的事情就都不用管了。持续集成工具就是以这样的思路来帮助集成工程师完成狭义集成工作的：持续集成工具一直盯着公共版本库，一旦检测到版本库里有新的提交，就自动调用源代码版本控制系统的相关命令，在最新源代码上打标签或者记录下自然版本号；然后依此自动创建或更新工作区；再自动调用编译命令、链接命令、打包命令；然后自动建立测试环境、自动部署、自动测试；最后自动发邮件报告集成是否成功完成，并确保安装包已经放到公共存储区里让大家都能访问到。

当然，如果在这个过程中，遇到了问题，那自然就需要人来干预了。但机器至少能做的是，记录下发生了什么样的错误，然后报告给你，通过电子邮件，

通过 MSN 或 QQ，甚至通过短信。嗯，这比枯坐在电脑前等待要强多了！

那集成工程师不就失业了？才不会。当集成出现错误、出现问题的时候，需要集成工程师来解决问题，或者由他来组织协调其他人来解决问题。这个工作有的时候相当不容易。另外，搭建和维护这一套持续集成解决方案，并且在每个项目中配置相应的参数和脚本，让它转起来，也需要人来完成。这些活儿才是真正值得用人类智慧来完成的。

> **[CruiseControl]** 开源世界里的老牌持续集成工具。提供了一个持续集成的框架。一般来说，稍作设置和补充，就能够实现上文所提到的自动化的持续集成。缩写 CC，有时候指的是它，有时候指的是 ClearCase[1]。
>
> **[Jenkins/Hudson]** 持续集成工具，新秀，开源。前些年叫 Hudson，最近因为 Oracle 跳出来索要商标权，所以只好改名叫 Jenkins。Jenkins/Hudson 和 CruiseControl 相比，操作界面更友好，配置更容易，运行更稳定。越来越多的项目开始使用 Jenkins/Hudson。

谈到持续集成工具，就不能不谈到构建管理工具。千万不要把构建管理工具误解成单纯的编译链接工具。构建管理工具和持续集成工具类似，提供了一个框架性的东西，这个框架适时去调用版本控制工具的命令，适时去调用编译链接工具的命令，适时去调用自动测试工具的命令，等等。这个框架掌控全局，发号施令，并收集各个工具的反馈和工作成果。

那么，构建管理工具和持续集成工具有啥区别呢？区别不大。它们的功能相近，只不过一个是从持续集成的角度去看待这摊事儿，所以给自己起名叫持续集成工具。一个是从构建相关的管理的角度看待这摊事儿，所以给自己起名叫构建管理工具。

另外，现在市面上的持续集成工具多是开源免费的；构建管理工具多是收费的，而且费用不菲。当然，在功能上，构建管理工具的功能更丰富些，对并

[1] 当然，CC 还可能指别的事情。在外企里，你大概会经常听到类似于“把那封邮件 CC 给张老三”之类的中英文夹杂的话，这里 CC 代表抄送。CC 还有可能表示立方厘米、云计算，等等。

行构建等方面的支持更强。

> **[ElectricCommander]** Electric Cloud 公司出品。就是出产神奇的构建加速器 ElectricAccelerator 的那个公司。
>
> **[BuildForge]** IBM Rational 出品。

那么，该选用构建管理工具还是持续集成工具呢？问题的关键是，你所在的企业，你所在的项目，对集成与构建相关的管理的实际需求是什么？持续集成工具在经过配置和少量定制后，是否已经能够基本满足你的关键需求？如果答案是肯定的，那么你很幸运，能省下一大笔银子，而且很有可能很快就能把工具用起来。

6.4　快！快！快！

如果能够快速完成提交前的质量保证工作，包括构建、测试以及走各种流程，等等，开发人员就会更乐于频繁地提交。相反，如果这个时间拖得比较长，开发人员就会倾向于攒上一堆工作，一起完成这些质量保证工作，一起提交。

而从狭义集成的角度来看，如果想频繁地集成，先决条件也是狭义集成过程要能比较快地完成，否则根本就频繁不起来。

那么，怎样才能更快呢？方法之一是，考察要完成的质量保证工作的内容，确定它们确实是必要的，并且必须是在这个阶段完成的。比如，在开发人员提交前，并不需要做全面的详尽的测试，只需要详尽地测试与自己改动相关的部分。而在狭义集成过程中，也同样不需要做全面的详尽的测试，只要进行粗略的系统级的测试，一般就可以了。全面的详尽的测试，留到基线产生以后，由系统测试团队完成。

测试团队也并非需要对所有的基线都做测试。只要挑拣些做就可以了。甚至，并非每轮测试都是全面的、详尽的。或者说，其中一些类型的测试要比另

一些类型的测试更频繁，而测试的详尽程度则较低。

这些是关于质量保证工作的内容的考量。为了让集成更快，还可以考虑采用一系列的技术手段，加速每项质量保证工作。以构建为例，可以考虑增量构建、提高机器以及网络的性能、防止用于构建的机器同时完成其他工作、使用多台机器并行构建，等等。这些都是我们在第 5.3 节和第 5.4 节讨论过的内容。

自动测试方面也类似。提高机器以及网络的性能、防止用于自动测试的机器同时完成其他工作、使用多台机器并行测试……静态程序分析也类似。

6.5 多层集成：当团队规模扩大以后

我们已多次提到，要经常提交，每个开发人员，完成一点工作后就应提交。同时，提交的内容应该具有了一定质量，不会对别人造成困扰。这两个要求，有时候是相互矛盾的。

假定张大侠负责模块 A，王大侠负责模块 B，李大侠负责模块 C。而其他大侠负责其他模块。现在要增加一个新功能，Spark，需要三人配合。模块 A 向外提供的一个函数叫 Color，有三个参数，标志着红、黄、蓝三色的值。现在，因为要增加新的功能，需要改变为四个参数，而含义则分别是色调、饱和度、亮度和闪烁频率。张大侠做完了修改，打算提交。但是认真负责的张大侠意识到，王大侠的模块 B 和李大侠的模块 C，是得做相应修改的。不然，程序就乱了套。所以，现在不能贸然提交。可是，张大侠一时半会儿又看不懂模块 B 和模块 C 的程序——本来嘛，是王大侠和李大侠负责的，只能让王大侠和李大侠改。可是，张大侠如果不提交，王大侠和李大侠就看不到张大侠新的函数接口，于是就改不了模块 B 和模块 C 中相应的调用它的代码。怎么办？

我们可以这么看问题。增加这个新功能 Spark，可以认为是一个大的任务。这个任务，拆成了三个子任务，分别交给张、王、李三侠做。只有三个子任务

都做完了，才能认为这个任务最终完成，达到一定的质量，可以提交。而如果在中途就提交，就可能会影响整个开发团队，让他们编译不通过，或者运行不起来。

从集成的角度看，这里面有两层集成。一层是，三个子任务集成为一个大任务，即增加一个新功能；而另一层是，这个大任务，与其他任务再集成。

在第一层里，虽然编译不通过，但是张大侠也必须提交——注意，这里只提交到张、王、李三侠所在的小圈子，不让其他人看见。而王大侠和李大侠在这一层的公共环境里，可以拿到张大侠刚刚提交的变更集。于是，王大侠和李大侠据此更新各自的工作区，并做相应修改，然后提交到这个小圈子。接着，张大侠临时扮演这个小圈子的集成工程师角色，测了测总体效果，嗯，没问题。再根据团队所有人的公共环境，更新这个小圈子的工作环境，然后再测测，嗯，还是没问题。于是，张大侠就把这三个变更集，一起提交到了团队所有人的公共环境里。这样，其他大侠再更新本地工作区的时候，就得到了这个新功能。

两层或两层以上的集成，被称为**多层集成**（Multilevel Integration）（见图 6-1）。

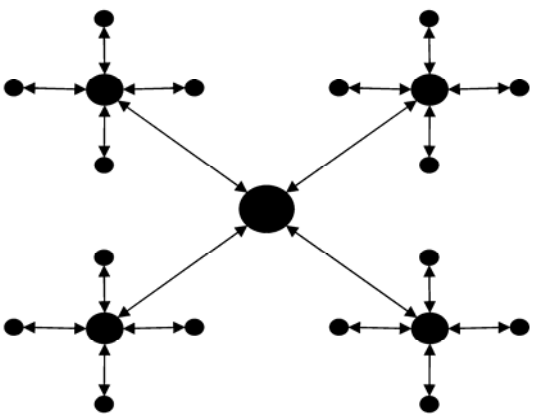


图 6-1　多层集成

什么时候使用多层集成？在上面的例子中，我们可以看到，张、王、李三

侠的修改，需要相互高度配合，不然连编译都通不过。这种情况，在小团队里可能偶尔会碰到。而团队规模越大，就会越经常碰到。原因是，团队越大，越倾向于有分工，不同的开发人员熟悉系统不同的部分，需要互相配合。并且，团队越大，系统越大，每一个新增功能或功能变化，所牵扯到的代码改动就越多，常常需要多人配合。

在上面的例子中，张、王、李三侠如果单独提交，那么连编译都通不过。而在实际工作中，使用多层集成时，常常还没到连编译都通不过这么严重的程度。也许能编译通过，只是运行不起来。也许程序能启动，只是某个相对重要的功能受到了限制。这些也同样会影响其他开发人员，或影响测试人员。比如，让他们感到困惑，误以为是 Bug。这些也值得考虑用多层集成的方法来避免。团队越大，系统越大，发生编译不通过、运行不起来等情况就会越频繁。

最终使用者也可能会因为程序中的半拉子功能而困惑。如果临近对外发布的时间，而又没有足够的把握在发布前完成某项功能或功能改动，那么应考虑在完成该任务后，一起提交。如果程序对外发布的频率很高，或者随时有可能发布，那么这一点就要特别注意。

当团队规模很大的时候，多层集成的另一个好处就会越来越明显，那就是，分担集成的工作，尽量在总的集成之前并行地解决更多的集成问题。假定集成有 100 个问题，其实 90 个都会在子集成的时候暴露出来，并被解决。而剩下 10 个，留给总的集成。这样，在子集成的时候，是几组间并行解决各自集成问题，只留下 10 个给将来串行解决。这比在将来串行解决这 100 个问题，要节约很多时间。不只是时间问题，是会不会变成一团乱麻的问题。我们已经讨论过对付一团乱麻的问题的方法了，那就是，尽早发现问题、分头尽早解决问题，不要等形成了一团乱麻再处理。在这里，这个原则也同样适用。

要想利用多层集成的这个好处，关键是如何分层，以及子集成之间如何划

分。有一个基本原则：内部高聚合，外部松耦合[2]。子集成所集成的内容，应该彼此高度相关。而它与其他子集成之间，则关系相对较弱。这样，有利于在子集成时尽量多地发现集成的问题，减少总体集成时发生的问题。比如在上面的例子中，就是一个大型任务内部，相互关联很强，于是先内部集成，再与其他部分集成。也可能是各个模块的修改先内部集成，再进行模块间的集成。

注意多层集成是有副作用的。简单地说，它走向了持续集成的相反方向，总体集成时，每次提交的代码改动量可能变大了。因此因分头开发而产生的问题可能会增多，开发人员间相互等待的情况可能会因此而增多，项目透明度可能会下降。

因此，多层集成应该只在确有必要的情况下再使用。在使用时，也要尽量减小总体集成时每次提交的代码改动量。某个大的功能，对客户来讲，必须是在要么没有要么全部实现之间二选一吗？是不是可以在即将到来的发行版本中先包括部分小功能？等下一个发行版本中，再包含更多的小功能？某个功能，如果在开发的过程中，尚未完成之时提交会影响测试，那么能不能建立起某种跟测试团队沟通的机制，预先告诉他们，某些部分先不用测试？

多层集成，在一些特定情况下是需要用的。那么，具体怎么实现多层集成？比如，第一层集成，发生在哪儿？工作区还是版本库？这个问题在下一章讲解分支时给出答案。

[2] 是不是听着很熟悉？上网查一查，这个原则本来是用在什么地方的。

7

第 7 章　分支：减少等待，分头工作

在自然界，几乎所有的树都有分支。分支能并行生长，并行地长叶子，扩大生长的空间，增加光合作用的面积，这不是仅仅一根树干所能挤得下的。

团队协作的本质是分头工作并且相互配合。在软件开发中，分支能起到这样的作用。合理地使用分支，能够实现并行工作，多头前进，适时汇合，从而减少等待和阻塞，大大加快项目的进程。

7.1　文件级分支

前面我们在讨论两个开发人员同时修改一处内容的时候已经粗略地谈到了分支，现在，让我们集中精力在这个话题上。

假定，出于不同的目的，张大侠和李大侠都要改 rsc.h 这个文件[1]。现在 rsc.h 的版本是 5。张大侠签出版本 5，基于这个版本修正 Bug A，签入，得到了版本 6。李大侠也是签出版本 5，基于这个版本，开发新功能 B，现在完成了一些重要内容，需要签入，以便保存一下。那么，应该保存为版本 7 吗？不，一定不能！因为这个新版本里，只含有为新功能 B 所作的改动，不含有对 Bug A 的修复。如果把它当成了版本 7，那么，版本 5 到版本 6 之间的差异，也就是对 Bug A 的修复，就丢了。

解决的办法之一是，李大侠签入的版本，命名为版本 5.B.1，意思是说，为新功能 B 的开发，我们开出一条**分支**（Branch），在该分支上保存第 1 个版本。

此后，李大侠继续开发，完成了功能 B。这时，他再次签入，形成了版本 5.B.2，意思是说，为新功能 B 的开发而产生的第 2 个版本。

此后，应该进行分支合并。否则，随后的开发人员就会困惑，到底是该跟着版本 6 后面开发，还是该跟着版本 5.B.2 后面开发？好像哪个都不对？在这里，分支合并的目的是，把分支上的改动，合并到**主干**（Main Branch）上来。主干又被称做主线、主分支、主控分支等，英文可以是 Mainline、Master Branch、Trunk 等。本质上，主干也是一个分支，只不过它特别重要，所以才有这些称呼。

合并的方法，具体地说，就是把从版本 5.B.2 到版本 5 之间的所有差异——包括从版本 5.B.1 到版本 5 之间的差异，和从版本 5.B.2 到版本 5.B.1 之间的差异——贴到版本 6 上面去。这样，在主干上形成了版本 7。

[1] 程序员出身的人都知道 rsc 是什么意思。这是一个暗号，让程序员辨清敌我。

合并是某种复制行为，合并不会影响到原分支，即那条叫 B 的分支。B 分支上已有的内容、已有的版本、已有的版本之间的差异，并不会随着合并而消失。它仍然在那儿，没有变化。

另一方面，合并这种复制行为，不是复制版本本身，而是复制版本之间的差异。因此合并并不会把 5.B.2 或者 5.B.1 本身带到主干上，而是把版本 5.B.2 到版本 5 之间的所有差异带到主干上，贴到版本 6 上面，形成了版本 7（见图 7-1）。

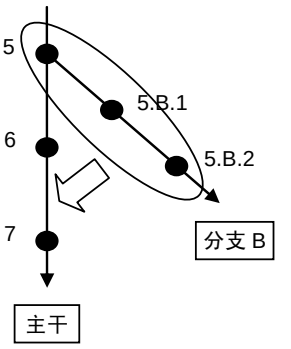


图 7-1　文件级分支

7.2　产品级分支

上一节，我们讲解了文件的分支，来引入分支这一概念。在文件层次上的分支概念，几乎在软件配置管理诞生之初就有了。历史的车轮滚滚向前，人们在使用分支的过程中，逐渐发现，其实被关注的，往往并不是具体某个文件的分支，而是整个产品的分支。是的，从现代的观念看，产品的分支比文件的分支重要得多。

当我们看待某一个文件的版本演进时，是这个文件的一个版本、又一个版本地形成的一条不断生长的线。而加入分支的概念后，就不再是一条线，而是不断分叉的线。主干上，可以产生分支，产生不止一条分支。每条分支上，可

以继续产生分支，可以继续产生不止一条分支。总体来看，就好像一棵倒着放的树[2]。每一条线上，即每一条分支上，有很多疙疙瘩瘩，也就是文件的版本。而最重要的疙疙瘩瘩，是在各个分叉的地方，那是一条或多条分支的起点。

与之对比一下，当我们看整个产品的版本演进而非单个文件的版本演进时，我们看到的是什么？我们看到的是，产品的所有源代码的一个断面，加上一个变更集，形成产品的下一个断面。有些断面被明确地标识：有个有意义的名字，常被称做基线。但多数断面默默无闻，顶多有个版本控制工具给它的自然编号。不论这些断面是被明确标识的，还是默默无闻的，它们都是整个产品的一个版本。产品的版本演进，是从一个断面，也就是产品的一个版本作为起点，贴上一个变更集，形成下一个版本，再贴上一个变更集，形成再下一个……产品的一个版本，跟着又一个版本，形成一条不断生长的线。而加入分支概念后，和文件的分支一样，就不再是一条线，而是不断分叉的线，好像一棵倒着放的树。每一条线上，即每一条分支上有很多疙疙瘩瘩，也就是产品的版本。相邻版本之间，是一个变更集。我们也可以把变更集看成是疙疙瘩瘩，而每完成一个变更集，就产生了产品的一个整体版本。

讲解文件的分支时，我们提到了合并。合并是把一条分支上的版本间的差异，带到另一条分支上，比如带到主干上。可能是把整个分支上的内容，从起点到终点间的全部差异，都带过去；也有可能，只是带其中的一部分，一些版本间的差异，一段一段的。与之相对应，产品的分支也同样。只是，明确地，一段代表着一个变更集。可以把一条分支上的全部变更集，都贴到另一条分支上去，也可以只是挑挑拣拣，把一些变更集，比如几个关于 Bug 修复的变更集，贴到另一条分支上去（见图 7-2）。

[2] 或者躺倒的树，或者立起来的树，或者树干在中心一点，枝条向四周发散的样子。看你打算怎么画了。

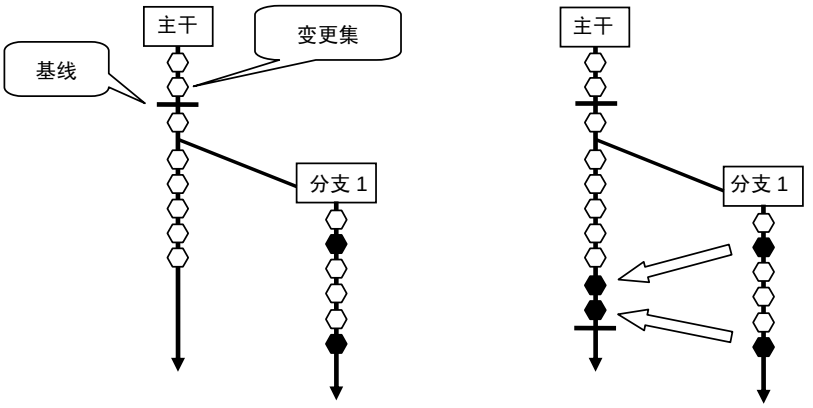


图 7-2　产品级分支

这里的合并，是某种复制行为，分支合并不会影响到原分支，变更集的复制也不会影响到原变更集。原分支上的变更集，以及原分支上的一个又一个整体版本，并不会随着合并而改变或消失。

类似于文件级分支，这里所说的产品级分支的合并是某种复制行为，也不是指复制分支上的某个整体版本，而是复制整体版本之间的差异，把这个差异复制到另一条分支；变更集的复制，不是指复制变更集所涉及的各个文件的修改后的版本到另一条分支，而是把变更集所涉及的各个文件修改前的版本和修改后的版本之间的差异，复制到另一条分支[3]。

如何实现产品级分支？不同的版本控制工具有不同的方法。有的工具用文件级分支模拟产品级分支。在这样的工具里，创建产品级分支，就是在相关文件上创建同名的文件级分支。有的工具则在架构上就是产品级分支的解决方案，并且在这样的工具里创建一个产品级分支的开销很小。

> [Subversion] 直接支持产品级分支。

[3] 只是有时候碰巧，刚好复制过去的文件的版本，就是原分支上变更集里该文件的修改后版本。如果原分支上变更集里，该文件的修改前版本的内容，与这条分支上，分支合并前，分支顶端的版本的内容完全相同，就会发生这样的碰巧。其实这样的碰巧，是经常发生的，但切不可因此而混淆了概念。

> **[Git]** 直接支持产品级分支。
>
> **[Base ClearCase]** 在 Base ClearCase 里，只有文件级分支。
>
> **[ClearCase UCM]** 在 ClearCase UCM 里，用流（Stream）来支持产品级分支。

产品级分支，在有些文章中也被称为码线（Codeline）。有些工具中，有些文章中提到的流（Stream）的概念，也大致对应于产品级分支。

在本书中，当提到源代码的分支时，如无特别说明，都是指产品级分支。

7.3　典型应用：实现多层集成

在第 6.5 节里，我们已经讲过了多层集成的概念：先把完成的任务提交到小圈子里，在小圈子里集成在一起，作为一个整体，再提交到整个项目团队的公共区域，在那里和更多的变更集相集成。我们当时没有讲如何实现这一点。现在我们来看看，如何借助分支来实现它。

“整个项目团队的公共区域”是主干，“小圈子”是一个分支。从主干上的某一点，也就是产品的某个整体版本开始，分支被创建出来，从这里开始生长。还记得么，张、王、李三侠，要一起完成一个叫 Spark 的新功能[4]。那么，我们在这里就把分支起名为 Spark。在 Spark 分支创建好后，张、王、李三侠把自己的工作区定位到这条分支上，而不再从主干取最新版本。

张大侠首先以一个变更集的形式向这条分支上提交了自己的修改，于是分支向前生长了一小节。分支的末端上，包含了张大侠对 Color 函数的函数接口和函数体的修改。王大侠据此更新自己的工作区，把分支的末端拿过来。李大侠也这么做了。显然，现在王大侠编译不过去了，李大侠也一样。但这个正是我们想要的，因为这说明，王大侠和李大侠能够看到张大侠的修改了。这样，王大侠和李大侠才能继续修改各自负责的模块里，对 Color 函数的调用。而另一方

[4] 就是第 6.5 节中的那个故事。

面，其他大侠，并没有工作在 Spark 分支上，也就看不到张大侠对 Color 函数的函数接口和函数体的修改，也就不会因此而编译不通过。

王大侠在自己的工作区里，修改了他所负责的 B 模块对该 Color 函数的调用，然后测了测，没问题，提交了。李大侠也在自己的工作区里，修改了他所负责的 C 模块对该 Color 函数的调用，也测了测，没问题，提交了。现在，Spark 分支又向前生长了两小节。

张大侠再次登场。作为实现 Spark 功能的技术负责人，也就是 Spark 分支的负责人，张大侠要把 Spark 功能最终放到主干上去。他首先比较全面地测试了一下 Spark 分支末端，Spark 这个功能完成的情况，以及改动可能影响到的其他功能。嗯，不错，只有一个小 bug，顺手就改了。接着，为了保险起见，张大侠又把主干上最近新增的内容合并到 Spark 分支。粗略测试一下，没问题！最后，张大侠把 Spark 分支上的改动合并到主干上。工作完成啦。

Spark 分支合并进主干后，其他开发人员就都可以通过更新自己的工作区到主干的末端看到张、王、李三侠为实现 Spark 功能所做的修改了。注意，这时看到的，是三个人所作的完整修改，所以是可以编译通过的，可以运行的，可以 Spark 的。

7.4 温故而知新：实现隔离与共享

上一节介绍了分支的一个典型应用，实现多层集成。是不是觉得有些东西似曾相识？没错，把 Spark 功能从 Spark 分支带到主干的过程，挺像是当初我们讲的，从工作区提交代码改动到版本库的过程。

我们讲过，提交前，要保证提交质量。相对应地，把 Spark 分支合并到主干之前，进行了测试。我们讲过，提交前，要更新工作区。相对应地，把 Spark 分支合并到主干之前，先把主干最近的增长合并到了 Spark 分支。最后，从工作区

提交到公共版本库。相对应地，把 Spark 分支合并到主干。当各位大侠从工作区提交改动到 Spark 分支时，用的是以前讲过的方法。当张大侠把 Spark 分支合并到主干时，用的是刚讲过的方法。

两种方法大同小异。因为它们背后是同样一个模式，就是通过一个位于中心的点来交换不同的开发人员或不同的开发子团队之间的代码改动。只不过，这个位于中心的点，可以是主干，也可以是分支。而位于四周的点，可以是工作区，也可以是分支。

位于中心的点，不论是主干，还是分支，都是在服务器端的存储库里。这意味着很安全，不必担心丢失。这意味着每个版本都被记录下来，可以回溯。这意味着在版本控制工具的支持下，不会发生版本覆盖的情况，不会混乱。

而位于四周的点，不论是用工作区，还是用分支，本质上都是达到了这样的效果：当使用它的开发人员不想受到来自外界的干扰时，他（们）不会受到意外的打扰，也不会打扰别人。工作区或分支是由开发人员完全掌控的，与外界隔离的。另一方面，当他（们）想与外界交互的时候，不论是把外界的东西拿过来，还是把自己的改动送出去，都有相应的方法去做。也就是说，分支和工作区一样，既支持隔离，也支持共享。

既支持隔离，又支持共享。有了这样的环境，开发人员就可以安全地、有效率地并行工作。不同的线索，并行地发展，并且在适当的时候相互共享内容。

如果你把“不同的线索”视为广义的分支，那么，工作区就可以视为广义分支的一种实现方法。版本控制工具中的分支，是广义分支的另一种实现方法。[5]

当然，工作区与版本控制工具中的分支相比，是明显的简化版。一般来说，工作区由如下两部分组成：某条分支（狭义分支，下同）上的某个节点，再加上在一些文件上的一些修改。这些修改还没有完成，没有提交，总的来说，是

[5] 分布式版本控制工具中，是用位于不同机器上的版本库，也可以实现广义分支。

一个未完成的变更集。因此，工作区的本质是，以产品的一个整体版本为起点，再加上一个未完成的变更集，就放在开发人员的机器上某个目录下。而相对应地，分支的本质是，一串已完成的变更集串起来，形成不断演进的整体版本，保存在版本库里。

因此，工作区与分支相比，有明显的功能上的不足。首先，它记录的容量小，只能容纳一个未完成的变更集，没法记录历史。其次，它的安全性低，开发人员的机器坏了，工作区就没了。最后，狭义分支上有版本控制，能防止版本覆盖，因此可以有多个人同时在它上面工作。而工作区只适合单个开发人员在此工作，否则有版本覆盖的风险。

因此，我们在一些情形下，需要使用版本控制工具中的分支，而不仅仅是工作区。

7.5 分支的其他典型应用

我们已经讲解了分支的一个典型应用：支持多层集成。本节我们将概要介绍其他典型应用。这些典型应用将在本书的其他章节中陆续展开。没错，分支非常重要，本章有限的篇幅是不够的。

- 典型应用二：管理交迭

举例来说，开发和发布补丁版本就是交迭的一种形式：1.0 版本发布上市后，在继续研发 2.0 版本的同时，打算基于 1.0 版推出 1.1 版，主要是修复了一些 1.0 版的 Bug，并进行了少量的功能增强。这时就需要分支来帮忙，从 1.0 版到 2.0 版的开发是在主干上，而从 1.0 版到 1.1 版的开发是在一条分支上的。

更进一步，在即将发布 1.0 版的时候，分支可能已经存在了。一条分支主要关注于提高程序质量到可以出售的程度，以便发布 1.0 版。而主干则更关注于增加新的功能，以便未来发布 2.0 版。

类似这些形式，本质都是交迭的，第 12 章我们会作详细的介绍。

- 典型应用三：支持变体

所谓变体，就是具有相似性，但是不完全相同的产品，或是产品的不同版本。比如，中文版和英文版，Windows 版和 UNIX/Linux 版。分支可以用于支持变体的开发。不同的变体在不同的分支上，而一般的、通用的本体则在主干上。一般的、通用的修改通常提交到主干上，然后同步到各个变体分支。而变体分支上，关于该变体的特殊的修改，通常不会合并回主干。第 13 章我们会作详细的介绍。

注意，支持变体的开发有多种方法，分支只是其中之一，并且常常不是最好用的方法。关于如何支持变体的开发，详见第 13、14、15 章。

- 典型应用四：隔离试探性工作

这个很好理解，当你不太确定如何实现某个功能的时候，需要做一些试探性的工作。如果工作量比较大，在工作过程中需要保存中间版本的话，那么单纯使用工作区就不够了，需要为试探性的工作建立分支，以保存中间版本。如果事后证明你的实现方法是可行的，那就把这条分支合并回主干。

所谓试探性的工作，也有可能是目标本身就是不确定的。这个功能真的是客户想要的吗？会不会有什么理解上的出入？还有的时候，开发人员的理解没问题，但当客户真正见到这个功能的时候，他会意识到这不是他真正想要的。如果这方面的风险比较高，那就先做个原型，帮客户找个感觉。这样的原型，如果完成得很粗糙，那么即使证明了它就是客户的需求，也不一定会合并回主干。

- 典型应用五：管理来自第三方的源代码

源代码除了由企业内部研发产生外，也有可能来自外部，可能购买自另一个厂商，也可能来自开源社区。另一个厂商或开源社区，对于企业来说都是第

三方。来自第三方的源代码需要妥善管理。直接访问第三方的版本库并在其上工作的想法当然不错，但在现实中可能有各种困难，比如权限的问题、访问速度的问题。常用的解决方案是，把来自第三方的源代码，一次性地或者多次导入到企业自己的版本库中。如果是多次，就需要分支来体现。

此外，企业可能基于第三方的源代码，根据自己的需要进行修改。这需要和纯输入型的源代码的各个版本区分开。因此需要另一条分支，反映自己的修改。而这两条线索之间，常常需要共享和交互：新的第三方的修改一般需要复制到自己的分支中去。而自己的修改，其中的一部分，出于各种原因，也得回馈给第三方。

相关详细讨论，见第 17 章。

7.6 公共版本库中的分支

在集中式版本控制工具中，只有服务器上有版本库，开发人员自己的机器上只有工作区。在分布式版本控制工具中，服务器上有公共版本库，开发人员自己的机器上除了工作区外，还有本地版本库。不论是哪一类版本控制工具，服务器端都有公共版本库，公共版本库里都有分支，需要管理。这是本节要讨论的内容。

当某个开发人员或集成工程师或其他人员为了某个目的希望到某条分支上工作时，他应该能够迅速找到那条分支，迅速定位到分支的名字。比如，开发人员想到为发布 3.0 版本而存在的分支上去看看某个 Bug 是否存在，他就需要知道那条分支叫什么名字，以便取得那条分支上的源代码，或者对应的安装包。当然，可以用某种列表的形式，列出每一条分支的名字，然后后面是对应的关于该分支的描述。然而，如果分支的名字本身就能够方便使用者理解分支的意图，那不是更直观，更一目了然，更便于查找？如果分支的名字足够有规律，使用者甚至能够直接猜到分支的名字，直接定位到那条分支上去。因此，分支

需要有意义的，符合规范的名字。

在分支命名规范中，通常会体现对分支的分类和分组。比如，所有的用于总体集成的分支，都使用 int 作为前缀（或后缀）。某个叫 aaa 的子团队用于子集成等工作的各条分支，都以 sub_aaa 作为前缀（或后缀）。这样做的好处是，即便是猜不出具体某条分支的名字，也能在很大程度上缩小查找的范围。

分类和分组的另一个好处是，便于划分和体现职责和权限。权限包括读权限、写权限和管理权限。读权限是指读取分支上特别是分支末端的源代码断面的权限。写权限是指向该分支提交和合并，使该分支继续生长的权限。管理权限包括分支的创建、重命名、删除等。对权限的限制，可以考虑通过版本控制工具提供的技术手段来实现。然而，这可能大大提高了解决方案的复杂度，也增加了不少管理成本。更为简单和实用的方法是：在分支命名上，让使用者清晰地意识到它属于哪类或哪组，并随之根据流程规定，甚至是根据常识，意识到自己有没有相关的权限，该不该进行相关的操作。这通常已经是不错的解决方法。

总之，由于服务器上版本库中的分支是公共的，因此需要有比较多的规范，需要有比较多的管理。并且，只在确实有必要的时候，再开新的分支，因为每条新的分支都意味着多了些操作和管理成本。

7.7　私有版本库中的分支

本节针对分布式版本控制工具。在分布式版本控制工具中，不仅在服务器上有一个公共版本库，每个开发人员在自己的机器上还有一个本地私有版本库。开发人员的工作区与这个私有版本库之间相互同步，私有版本库和服务器上的公共版本库之间相互同步。

服务器上的版本库里的分支，一般来说，都会在本地的版本库中反映出来。

我们称它们为本地版本库中的**远程分支**（Remote Branch）。远程分支是远程（服务器上）版本库中的分支在本地版本库中的镜像，是只读的。而在本地版本库中的修改，发生在**本地分支**（Local Branch）上。当提交本地分支上的修改到服务器时，通常是提交本地分支上的变更集到服务器版本库中的分支上（比如说主干上），而不是复制本地分支本身到服务器上的版本库里，让公共版本库里多出一条分支来。因此，本地分支，通常只在这台机器在这个版本库中存在，是私有的。

本地分支，是这个开发人员（或集成工程师）的“私事”。而且，其数量跟公共版本库中的分支数量相比，一般来说也少得多。因此，怎么命名、怎么管理，一般来说是这个开发人员凭常识，觉得怎么对劲儿怎么来，就可以了。

对本地分支上的每个整体版本的质量的要求，也大大下降。只要提交到服务器的最终版本是高质量的，中间版本是什么样的，别人不太关心。事实上，能够使用本地分支，方便地存储中间版本，是分布式版本控制系统的强大功能之一。

由于能够方便地存储中间版本，开发人员就还可以方便地放下尚未完成的某项工作 A，去完成另一件更紧急的事 B，然后再回到 A 上继续工作。比如下面这种情形：张大侠正在潜心重构一个模块，已经工作了一整天，大概还需要一两天的时间。这时候，王大侠和李大侠心急火燎地找来了，因为张大侠负责的另一个模块有一个要命的 Bug，导致王大侠和李大侠没法继续开发，这个 Bug 得赶快修复。

于是，张大侠先把未完成的重构工作提交到一条本地分支上，然后建一条新的本地分支，修复 Bug，提交到本地版本库，再同步到服务器上的公共版本库。这件紧急的事情办完了，张大侠再从以前的那条本地分支上，把当初未完成的重构工作拣回来到工作区，继续工作。等工作完成，提交到本地版本库，再同步到服务器上的公共版本库……

这种情况下，变更集是一个尚未完成的任务，或完成质量尚不高的任务。不必担心因此把版本库中的历史记录弄得不好看，将来影响其他开发人员理解。分布式版本控制工具通常提供了改变历史的功能，改变本地分支上的记录。趁着修改还没有同步到服务器端的版本库，开发人员可以改变本地版本库中的历史记录。比如，把一些增补的修改塞到已经提交到本地版本库的变更集中，或者把本地版本库中的几个变更集合并成一个变更集。做完这些事，本地版本库里的历史记录变得清清爽爽。将来把变更集同步到公共版本库，那里的历史记录也清清爽爽。

在集中式版本控制系统中，修改版本库的历史记录是一件“恐怖”的事情，因为版本库是公共的，存储的数据和信息是供多人使用的。在这种情况下，修改历史，很容易引起混乱，安全性也有很大隐患。在分布式版本控制工具中就不同了，“我的地盘我做主”，因为跟别人没关系。

> **[Git]** 就是这么玩儿的。

由于在分布式版本控制工具中，本地分支“便宜”、好用又安全，所以人们在使用时倾向于为每一个新的工作任务都开一条新的本地分支，这条分支通常基于远程分支。这样做，是为了去除不同的工作任务间没有必要的依赖。比如，当你完成了一个工作任务对应的变更集后，可以放在一边，等其他同事来做同行评审（并且你可能根据同行评审的结果稍作修改）。而同时，你已经可以开始为新的工作任务，在新的本地分支上工作了。两条本地分支，不会相互依赖。版本树也很清晰，将来每个工作任务，通常只反映为公共分支上的一个变更集。

8

第 8 章　管理文档

软件配置管理，是关于软件资产的管理，是关于演进中的软件资产的管理。我们已经花了很多时间谈论源代码的管理，还花了一点时间谈论安装包的管理。下面，我们来谈论文档的管理。

8.1　文档的标识和存储

文档与源代码的本质区别是，源代码是人编出来给机器看的，而文档是人写出来给人看的。在一个软件研发项目中，会产生各种各样的文档。这些文档以不同的形式存在。源代码文件里的注释信息，本质上就是文档，因为它是给人看的。如果注释信息比较多，就可能会剥离出来，作为单独的文本文件。这时候，就有点像是我们通常概念上的文档了。如果这样的文档与产品的某个局部紧密相关，与产品在源代码实现上的某个细节紧密相关，如果主要是开发人员阅读它，那么，将它和源代码一同管理比较好[1]。

而我们通常所说的文档，是篇幅比较长的，格式比较复杂的，带插图的，项目、产品或组件一级的，看起来比较正式的文档，通常不仅是开发人员在读。这类文档，通常不再是一个简单文本（Plain Text）文件，而是用 MS Word、MS Excel 等写成的文档；这类文档，通常被单独管理，因为它们与源代码相比，有所不同。

首先，文档通常“文责自负”，通常有明确的所有者。一般来说，只有他会改动该文件。其他人改这个文件的可能性不大。而当其他人确实需要改动该文件的时候，通常会先告诉文件的所有者一声，避免因同时修改而产生版本覆盖。如果觉得这样还不够安全，希望由版本控制工具提供支持，那么通常是使用工具提供的锁机制，以保证不同的人串行修改同一个文档，而非鼓励大家并行修改之后合并。这是因为，如果两个人同时修改同一个文档，那么将来合并时，通常会有一定的技术困难和时间上的不少耗费。

其次，文档通常“独立成篇”，文档之间相对独立。为某个原因修改源代码的时候，常常需要修改几个相关的源文件，然后一同编译、链接。而为某个原

[1] 需要与源代码保持紧密一致的文档，也值得考虑与源代码一同存放、一同管理。典型的例子是用户使用说明文档。当然，也有其他一些手段，用以保证文档与源代码的一致性。

因修改文档的时候，常常是只需要修改一个文档。在修改之后，也不存在一起编译之类的问题。因此，为获得整体环境而创建工作区，意义就不大了。用变更集来标识对多个文件的修改，意义也不大[2]。文档如此独立，阅读者如此广泛，以至于经常被通过各种途径传播，比如 E-mail 等。因此，文档自己说明自己的能力，就变得重要[3]。这是下一节的主题。

最后，文档与文档之间，文档与源代码之间，经常是不一致的、不同步的。需求文档里写的需求，可能还没有转化成设计。设计文档里提到的结构，可能还没有转化成实际代码、实际功能。机械地要求它们始终一致，只会降低工作效率。通常它们都是以一个对外发布版本或一次迭代结束时的版本为目标，争取在那时，该版本对应的需求描述和程序实际功能是对应的。

因为上述原因，这类文档被单独管理，而不是与源代码放在一起。

下面我们看一下，如何存储这些文档。可以考虑使用以管理源代码为主要目的的版本控制工具来存储文档。但是如刚才所述，很多功能都会用不上，比如工作区、变更集。也有的功能会不好用，比如文件的合并[4]。因此，可以像对安装包的管理一样，通常用可以共享访问的文件系统来存储文档，把文档放在共享目录里，就基本够用了。

不论是用特别的工具来存储文档，还是简单地把文档放在共享目录里，都要考虑目录结构的合理规划，比如用什么样的结构，叫什么名字；要考虑文档文件的合理规划，比如要有哪些文档，存放在哪个目录下，文档叫什么名字。这些规划，不仅是每个项目、每个产品的规划要合理，还应该考虑，在一个项目中积累的经验，拿到下一个项目中使用。这通常体现为目录结构的模板，其

[2] 尽管文档之间相对独立，有的时候，标识几个文档文件的整体版本，仍然是有意义、有价值的。甚至，标识文档与源程序之间的版本跟踪关系，也是有意义、有价值的。

[3] 事情是相辅相成的。由于文档通常具有自我描述的能力，并且文档通常有明确的所有者，因此对它可以管得比较松，允许通过 E-mail 等方式传播，而非一定要从版本控制工具里取得。

[4] 有些文档编写工具本身具有合并文档的功能，比如 MS Word。而一些版本控制工具可以调用文档编写工具的功能来实现合并。

中含有各个文档的模板，以此作为下一个项目的起点。

除了考虑存储结构外，还应该加以一定的权限管理，规定谁可以读，谁可以写。

每个文档文件的命名，除了文档名称本身外，一般应加版本号，共同组成文档文件的名称。同一份文档的不同历史版本，是对应于若干个文件存储在一起的。这些文件，是通过版本号来相互区别的。

除使用共享目录来管理文档外，还可以考虑使用针对文档管理特点的，具有版本控制功能的文档管理工具来管理文档[5]。

> **[SharePoint]** 微软的 SharePoint，至少是一个文档管理系统。此外，它还扩展了 Exchange Server；还提供了可扩展的协同工作解决方案；还支持网站建设，是高度可定制的；还扩展了 InfoPath，一种用于集成表格管理和数据传输的桌面应用程序技术……

8.2　自带的说明信息

如上节所言，文档通常有明确的所有者，由他修改，其他人只是阅读。文档通常独立成篇，文档间的耦合性不强。因此，与源代码不同，文档的传递和复制，通常管制比较松。这就要求，文档本身的一些信息，比如版本信息等，要由它自身携带。

最简单的携带方法，是通过文件名。文件名包括文档名称和版本号两部分，但这常常是不够的。文档中起始的一到两页，常常用来更全面地记录和展现相关信息。我们来看看这包括哪些部分。

关于文档的名称，在文件名中，可能只是以缩写体现的。在文档中，就应该给出文档名称的全名。比如，文件名中的缩写是 SCMP，在文档起始页中，应

[5] 当然，文档管理工具的功能不仅限于版本控制。比如，它也可以记录第 8.2 节中的信息，并依此支持相关流程的执行。

给出全名——软件配置管理计划，最好再给出英文名——Software Configuration Management Plan，不然别人会弄不清 SCMP 这个缩写的含义。

应该以某种方式指出文档的“出处”。就好像告诉别人地址的时候，除了说“我家住 2 单元 301 室”之外，还应该告诉别人，哪个小区、几号楼。甚至，哪个城区，哪条街道；甚至，哪个国家，哪个城市[6]。这取决于，谁是可能的听众。文档也一样，或许项目之外的人也会看到这个文档；或许读这个文档的人，在参与不同的项目；或许很久很久以后，考古工作者发现了这个文档。所以，文档至少应该标识出所在的项目或产品。如果是模块级的文档，那应该标识出所在的模块。如果做得更好，还应该标识出是哪个公司的文档。对于大型企业，还应该指出是属于哪个分支机构的。这样标识，也为寻找该文档的“官方”存储位置提供了线索。有时候，我们看到一份文档，会关心它有没有更新的版本，而这要到文档的“官方”存储位置去找。

与此相关的，是版权信息和关于保密的信息。文档通常会有一定的密级，不同的文档，甚至是同一文档在不同的时期，密级可能不同。应该在文档起始页明确标出文档的密级，让使用者留意自己的行为。这是公司法务部门特别关注的，他们同样关注版权信息。这通常也明确标识在文档起始页。

关于当前版本的信息，主要有以下三项。首先是版本号，其次是版本产生时间，再次是版本的目的或状态。源代码整体版本是有目的和质量状态的，对于文档也类似。只是，通常单个文档就有其状态，因为它已经独立成篇了。文档的状态通常有如下三种：**草稿**（Draft）、**评审中**（In Review）和**已发布**（Released）。草稿状态是正在撰写过程中的状态；评审中是指草稿写完，送交相关各方评审；已发布状态是指评审通过，领导认为可以正式发布它，供文档的使用者使用。

文档的作者和参与者的信息，也应该被记录。比较全面的记录内容包括：

- 谁是这份文档的主人，对文档的内容及文档书写的协调工作负责。

[6] 一般来说，不需要比这更高一级的标识了。

- 哪些人是这份文档的主要作者，书写了这份文档的绝大部分内容。对于中小型文档，通常主要作者只有一个人，且他就是文档的主人。
- 哪些人为这份文档做出了贡献，包括正式的评审和非正式的建议等。
- 如果这份文档是已经被批准，正式发布的文档，那么，是哪些人批准的，以谁的名义发布的。

以上四点，应酌情考虑记录。

关于版本历史信息。每次修改并保存版本，通常要记录如下信息：

- 修改后的版本号。
- 修改完成的时间。
- 本次修改的修改者。
- 这个版本的最终状态。
- 对本次修改的简单描述。

8.3　趋势：Wiki

用共享目录下的众多文件来组织和保存文档，有一些不便之处。比如，对文档的并行修改；比如，查看文档修改的历史记录；比如，如果想知道关注的特定内容在哪个文档里，就得依次打开各个文档，看其简要介绍。有没有什么办法，能够去除这些不便之处？有没有更好的组织和保存方式？Wiki 应运而生。

Wiki 一词来源于夏威夷语“wee kee wee kee”，是“快点快点”的意思。中文译为“维客”或“维基”。Wiki 首先是一种语言，一种像 HTML 一样的既有内容，又有格式的语言。而与 HTML 比起来，用 Wiki 写东西比较快。它比 HTML 要简单得多，学起来也容易得多，不比写 E-mail 难多少，可是能表现的内容却像 HTML 一样丰富。使用 Wiki 语言写作的方法是，当你编辑网页的时候，在 Web 浏览器里，登录 Wiki 系统，在特定页面的可编辑区域里，用 Wiki 语言书

写、修改和保存。而在展现给别人（和自己）的时候，Wiki 系统自动把它翻译成了 HTML 语言，在浏览器中展示。大家像浏览普通页面一样，浏览 Wiki 语言写成的内容。

Wiki 的修改和展示都是基于网络的，都是在浏览器里完成的。所以，不需要为使用 Wiki 而在本机安装什么特别的应用程序。另一方面，不论浏览不同页面，还是执行修改操作，都只需要点一两下鼠标就能完成，用 Wiki 很方便。

Wiki 是通过 Web 页面展现的，通过 Web 页面组织的。这意味着，通过超链接，不同页面间的相互引用要比文档文件内部的不同章节相互引用方便容易得多，更比不同文档文件之间的相互引用方便容易得多。通过超链接，使一份文档的内容，以更为友好的方式带领人阅读，特别是查阅。不仅能知道本页引用了哪些页，还能通过 Wiki 提供的功能，方便地查到哪些页引用了本页。通过超链接，使不同的文档，以更有意义、更容易理解的方式组织到一起展现，而非仅仅是一个共享目录名下的若干文件名。

快、容易、友好，只是 Wiki 特点的一部分。更重要的是，它有内在的版本控制机制。在你修改前，需要进行类似于签出的操作——单击“编辑”按钮。在你修改完成后，需要进行类似于签入的操作——单击“保存”按钮。于是，它就会记录修改的历史：各历史版本，谁修改的，修改了哪些内容，为什么修改等。同时，它还有相应的机制，防止版本覆盖、修改内容丢失。另外，可以设置相应的权限控制。这么看来，Wiki 是某种版本控制工具。不同的是，传统的版本控制工具，最小的控制单元是文件。而 Wiki 最小的控制单元是页面，甚至是一个页面的某一部分。

在 Wiki 页面中可以嵌入附件，比如 MS Word 文档、MS Excel 报表等。Wiki 对附件同样有版本控制。附件的每一个版本都会被记录，连同相关的信息：谁上传的，何时上传的等。

Wiki 提供的搜索功能，也比传统的版本控制工具强大。传统的版本控制工具顶多提供对文件、目录名称的搜索。而 Wiki，提供的是对内容的搜索，即全文检索，这有点像是广域网上的搜索引擎。这对于查阅文档来说特别实用。

什么类型的文档适合于用 Wiki？通常企业内部信息比较适合用 Wiki 来承载。比如过程和流程信息、工具安装指导、使用技巧和常见问题解答、组织结构与联系人信息、项目基本信息及当前关注要点等。这些信息要便于查阅，便于更新，而这正是 Wiki 所擅长的。

事实上，这些企业内部信息是**知识管理**（Knowledge Management）的典型管理对象，而 Wiki 是知识管理所用的典型工具。

> [MediaWiki] MediaWiki 是全球最著名的开源 Wiki 引擎。它是维基百科全书的系统软件，并有大量其他应用的实例。它对中文的支持也很不错。
>
> [其他 Wiki 系统] 其他 Wiki 系统还有很多，比如老牌的 Twiki、小巧的 DocuWiki 和 Wikka Wiki。它们各有优缺点，你在网上能找到很多它们之间相互比较的帖子。

8.4　趋势：数据文件和数据库

文档的本质是给人看的信息。给人看的信息，用章节段落的形式组织，用文件来承载，不一定总是最方便的。

举例来说，比如需求文档。需求常常是有层级的，产品级的需求是由子产品级的需求实现的，产品级的需求与模块级的需求之间有复杂的关系。每一条产品级需求常常需要若干个模块提供的功能相互配合，才能实现，也就是说，每一条产品级需求，常常对应于多个模块级的需求。另一方面，模块提供的某个功能，可能会参与多个产品级需求的实现，也就是说，每一条模块级需求，常常对应于多个产品级的需求。在这样复杂的关系里，要做到需求不丢、不重复，可不是件容易的事。把需求写进一个文档或写进若干文档，在需求间互相参考、互相对照，常常比较吃力。而需求有变化调整的时候，需要考虑相关需求的变化调

整，这也同样不容易。而如果用文档来记录需求，那么会有一个更大的不便：需求一旦变化，就改写文档吗？那可能文档会有数百个甚至上千个历史版本……

更好的方法是用一套需求管理工具来管理需求信息。每个需求是一个条目，条目可以有各种属性；条目有它的修改历史；条目之间的关系被记录，一旦一个条目被修改，相关的条目会被提醒要不要修改等。需求管理工具把这些数据，保存在特定格式的数据文件里，或者数据库里。使用需求管理工具来记录和管理需求，比用一般文档文件来记录和管理需求，要方便得多。

不只是需求管理工具，系统建模工具也一样。它能用图形的方式，从不同的角度对产品建模。同时，它可以表达和维护产品不同模型或模型的元素之间的联系。显然，这是一份 MS Word 文档文件很难做到的。系统建模工具，同样把这些数据保存在特定格式的数据文件里，或者数据库里。

不只是需求管理工具和系统建模工具，还有其他的一些工具，为软件开发活动提供支持，这些工具都将保存和使用数据。此外，不同类型的信息，也可能需要记录其间关联关系。比如，有些企业将需求条目与对应的测试用例关联起来，通过管理需求条目的系统与管理测试的系统集成，自动生成测试用例。

所有这些数据和信息，和源代码一样，都是软件项目的资产，同样应该进行软件配置管理。这主要通过两种途径实现。

途径之一是，由相关工具本身提供一定的软件配置管理功能。比如，需求管理工具会记录每一个需求项的历史，记录谁、何时、做过什么样的改动。这些“数据的数据”，与需求描述本身，被记录在数据文件或数据库里。这有点像传统文档：传统文档把“文档的文档”放在文档的起始页。但是在这里，工具所记录的要细致得多——可以针对每一项或每一个元素；也自动化得多——谁改动的、在何时改动等信息，通常是自动记录的。

途径之二是，对数据文件或数据库，进行外在的、整体的管理。考虑数据

文件与数据库的备份，通过一定的备份机制，保证当数据的存储发生问题时，可以得到尽可能的恢复。考虑记录数据文件的不同版本及历史信息，或简单地保存数据文件的各个历史版本，作为不同名称的文件。数据库所涉及的一个或多个数据库文件，考虑作为整体来保存，保证可以恢复到历史上的某个断面。这样，保证当数据的演进发生问题的时候，可以参考或回退到历史上的整体版本。不论采用版本控制工具，还是基于普通文件系统，都要实现上述对数据文件和数据库的版本管理。

途径一和途径二常常是相辅相成的。途径一更关注一个数据文件或数据库内部发生的事情，而途径二更关注整体，而这两方面都需要。比如在权限设置方面，在工具内部，设置谁可以访问哪个特定部分，进行什么样的特定操作；而在工具外部，设置谁有权打开、修改、重命名、删除这个数据文件或数据库。

9

第 9 章　跟踪缺陷，直到消灭

至此为止，我们一直在谈论软件资产和软件资产的改动，谈论如何管理它们。当我们谈论软件资产的改动的时候，比如谈论 Bug 修复的时候，我们关注于具体改了哪个文件，哪几行，从哪个版本改到哪个版本。但是，这并不是源头，源头是在 Bug 产生的时候。这也不是终点，终点是在测试团队验证这个 Bug 确实被消灭的时候。在这一章里，我们关心一只虫子[1]出生入死的整个生命周期。严格地讲，是很多只虫子。

[1] Bug，英文原意是小虫子。

9.1　别让发现的缺陷跑掉

假定只有一个开发人员，比如就是你；假定只有一个用户，比如老赵。现在来看看你们之间关于 Bug 方面的交互。每当你给老赵程序的一个新版本，老赵就开始试着用它。由于经常出现问题，他会经常来打扰你。于是你厌烦了，跟他约法三章，“每天下午四点，我们讨论你发现的 Bug！”

第一天，老赵发现了两个 Bug。下午四点，向你汇报。哈，这两个都是小问题！老赵说的时候，你就顺手改好了！

第二天，老赵发现了 10 个 Bug。下午四点，向你汇报，汇报了 7 个。因为他没全记住。你一边听，一边翻白眼，因为好像有几个解决起来有点困难…… 等明天上午，好好想想。

第三天，上午，你在改 Bug。改好了 3 个。因为你忘了 3 个。还有 1 个，老赵说不清楚，让他重新演示一下，可这次又没出现异常情况。你把改好后的程序的最新版本给老赵，老赵欣然接受，但是没等一根烟的功夫，他就嚷嚷起来了。

为什么会发生这样的问题？因为这些 Bug 没有记录下来。其实只要对此做一点改进，就能基本解决问题了：老赵把 10 个 Bug 写在纸上。在下午四点，你们一起过一遍这张纸上的内容，然后你把这张纸放在电脑旁边[2]，直到 Bug 都被解决。这张纸，就是最简单的 Bug 跟踪系统。

Bug 跟踪系统，学名叫**缺陷跟踪系统**（Defect Tracking System）。据此推算，Bug 的学名叫**缺陷**（Defect）。本书自此处开始，改称它为缺陷。

通常使用的缺陷跟踪系统当然比上面那个复杂得多，因为实际情况常常复

[2] 其实老赵在把小纸条交给你之前，自己偷偷复印了一张。这只老狐狸。

杂得多。不只一个开发人员，可能是一大帮；不只一个用户，可能来自五湖四海。恐怕不只是用户直接找开发人员，还有客服人员、产品经理等在其中。另外，缺陷也不光是被用户发现，事实上，相当多的缺陷是被测试团队发现的。

有些开发团队使用 MS Excel 等电子表格来处理缺陷跟踪，这比用小纸条先进了不少。但是，通常仍然是不够方便的。比如，若两个人同时修改一个 MS Excel 文件，要小心发生修改丢失的情况[3]。再如，如果缺陷数量多了，用一个表格来浏览，也会很不方便。对于某一个开发人员来讲，可能没几个缺陷是归他负责的，可是却要为此在成百上千个缺陷条目里翻腾，找自己的名字。

基本的建议是，安装并使用缺陷跟踪软件。在这样的软件里，每一个缺陷是一个条目。缺陷被发现的时候，就被录入到这样的软件里。之后，软件帮助我们跟踪它，跟踪一个缺陷的整个生命周期，直到确认这个缺陷已经被消灭。

> **[Bugzilla]** 这是开源世界老牌的缺陷跟踪软件。类似的开源软件还有 Redmine、Mantis、Trac 等。
>
> **[BugFree]** 这是中国人自己的缺陷跟踪软件，开源，简单，用户广泛。最近它成为禅道项目管理软件的一部分。
>
> **[ClearQuest]** IBM Rational 公司出品。功能强大同时价格不菲的商业版缺陷跟踪软件。类似的商用软件还有 Telelogic Change 等。
>
> **[JIRA]** 这几年 JIRA 变得越来越热门。开源社区可以免费开源地使用它。而如果你的公司用它，那就需要付很少的一点钱，然后也可以得到它的源代码，并二次开发。以上是它的商业策略。在功能方面，大家对它的评价是“功能全面，界面友好，安装简单，配置灵活，权限管理以及可扩展性方面都十分出色”。事实上，它不仅是缺陷跟踪工具，也是项目管理工具。

9.2 状态转换图

烤鸭是如何制作出来的？首先，捉一只鸭子——鸭子的状态是被抓到。然后，结束它的生命——鸭子的状态是被杀死。接着是处理它——煺毛、打气、掏膛、

[3] Excel 的共享编辑模式，有助于克服这个问题。

洗膛、挂钩、烫皮、打糖、晾皮和灌水。这时，鸭子的状态是处理完毕。然后是烤制，右背侧、左背侧、左体侧、右体侧、鸭背，循环往复，直至烤熟——这时，鸭子的状态是烤熟。最后就是片鸭了，片成 108 片儿，每片儿都要有肉有皮——这时，鸭子的状态是片好。制作完毕。

这跟缺陷管理有什么关系？有的，制作烤鸭是一系列工序，这些工作可能是由不同的人完成的，鸭子在这些人手上传递。同时，这些工序让鸭子到达了一个又一个新的状态，直到最后可以端盘上桌。对缺陷的管理也一样。消灭缺陷也是一系列工序，这些工作可能是由不同的人完成的，缺陷在这些人手上传递。同时，这些工序让缺陷到达了一个又一个新的状态，直到最后确认已经消灭。让我们来具体看一看。

第一道工序一定是发现了这个缺陷，并且报上来。这时候，要在缺陷跟踪系统里填张单子。一般是谁发现谁填，但是如果用户不想填的话，就要客服人员代劳了[4]。填好了这张单子，就是在缺陷跟踪系统里创建了一个条目，代表这个缺陷，并且记录下是谁填的。好了，从此以后，这个缺陷就被盯上了。这时候，这个条目的状态，或者被说成这个缺陷的状态，是**已报告**（Reported），又称已创建（Created），被打开（Opened），初始（Initial），等等。

第二道工序是大致看看这个缺陷，并分配给合适的开发人员去处理。并不是所有的缺陷条目都真的是缺陷，有的可能是用户误操作；并不是所有的缺陷条目都是不重复的，可能不同的条目反映的是相同的缺陷；并不是所有的缺陷都需要立刻修复，得先处理比较着急、比较重要的。对于缺陷，要分配给合适的开发人员。合适的人选是，比较熟悉这个缺陷所在的那个模块，并且现在不是太忙的。一旦分配决定了，在缺陷条目里填上开发人员的名字，这道工序就算完成。这个时候，这个缺陷条目的状态就是**已分配**（Assigned）。

[4] 注意，用户和客服人员所报告的、所跟踪的，不仅是软件产品的缺陷，还可能有关于服务、生产、交付方面的抱怨等。准确地讲，他们是在用问题跟踪系统，而不是严格意义上的缺陷跟踪系统。问题跟踪系统的概念将在第 10.1 节介绍。

第三道工序是解决这个缺陷，这道工序由开发人员来完成。一般来说，会由一个开发人员创建一个变更集，对应于解决这个缺陷。不过，有的时候，需要多个开发人员配合工作，用多个变更集共同完成对这个缺陷的修复。当然，还有的时候，改了一处地方，几个缺陷都消失了。当开发人员解决了这个缺陷，并且提交了相关的修改，这道工序就完成了。这个时候，这个缺陷条目的状态就是**已解决**（Resolved）。

第四道工序是确定这个缺陷真的被修复了，这道工序一般由测试人员完成。集成工程师集成了各位开发人员提交的缺陷修复的相关代码，形成了新的版本。在编译、链接、打包和粗略测试通过后，集成工程师把安装包交给测试团队。测试团队逐个研究那些据称刚刚被解决的缺陷，进行测试，看看它们在新的程序版本里是不是真的消失了。如果真是这样，那么测试人员就放心了。于是，最后一道工序完成了。这个时候，这个缺陷条目的状态就是**已关闭**（Closed）。图 9-1 就是缺陷的状态转换图。

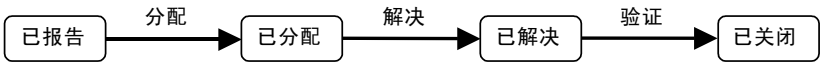


图 9-1　状态转换图（主流）

这只是主流，只是一般过程。在缺陷的一生里，难免发生意外。比如，我们已经提到，处于已报告状态的缺陷条目可能并不是真正的缺陷，那么，应该直接把它关掉，让它的状态转变为已关闭，当然，同时加上些注释说明一下会比较好。同样，对于重复的缺陷条目，也把它关闭，并且加上注释，说它跟某某条缺陷是重复的。而状态是缺陷条目已解决，可能并未真正解决，测试人员发现了这一点，把它重新打开，然后通常会直接分配给上次修复它的开发人员[5]，于是，这时候缺陷的状态变回已分配。考虑到这些因素，应将缺陷的状态转换图修正为图 9-2 的模样。

[5] 缺陷跟踪系统会自动记录下这一点，记录下这个缺陷没有一次就改好。将来领导是能查到的。

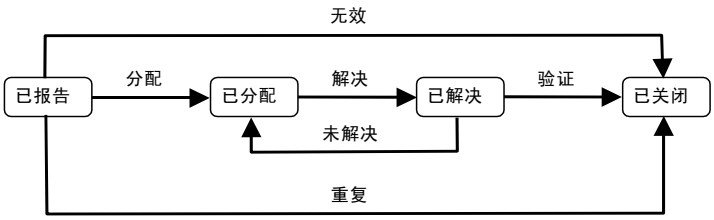


图 9-2　状态转换图（完整）

值得一提的是，不同的组织、不同的项目，可能会定义不同的缺陷处理流程。有时候会把其中的一道工序划分成多个小工序，就好像把处理鸭子这道工序，分成褪毛、打气、掏膛等若干道小工序，每一道工序完成后，都可以有对应的状态。还有的时候，把处理过程本身也设为状态，比如正在解决、正在评估等[6]。所以，状态转换图也会有一些差别。不过原理上都差不多。

由于为缺陷设置了状态，我们就能知道下一步该对它做些什么。并且，也便于查找：当项目经理关心还有哪些新缺陷没有被分配的时候，当开发人员关心有哪些分给自己的缺陷还没有处理的时候，缺陷的状态就特别有用。

> **[Bugzilla 等开源工具]** 支持状态转换，有默认的解决方案。如果想改变方案，可能需要阅读其源代码，并加以修改[7]。
>
> **[ClearQuest]** 支持状态转换，有默认的解决方案。并且方案本身能够方便地在图形化的界面上修改或者重新设计。

在本质上，我们是在讲述**有限状态机**（Finite-State Machine，FSM）在缺陷管理中的应用。有限状态机是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型，编程序的人都知道。

[6] 还有其他的情况。比如，当已经关闭了的缺陷，在几个版本之后再次出现时，就应把它的状态从“已关闭”转到“重新打开”。
[7] 哦，别担心，不是那么难。

9.3 准确记录，便于修复

我们已经讲过了，缺陷必须被跟踪，以防止丢失，以防止忘了解决。缺陷要有状态，不同状态间的变换，体现了缺陷被解决的过程。是的，记录缺陷，管理缺陷，关键是要解决它。而为了解决它，必须先把它描述清楚，便于找到合适的人来解决，也便于解决它的人发现问题究竟出在哪儿。

每条缺陷记录有一行标题（Title）或者叫总结（Summary），用来简述缺陷的内容。这一行一定要有所概括，又不要太抽象。比如，“飞船在高速行驶的情况下，会产生颠簸”就是个好标题；而“飞船在以大约 0.2 倍光速行驶的情况下，会忽然颠一下，有时候连续颠两下，有时候连续颠三下，之后会保持一段时间平静，大约 5 到 10 分钟后，重复颠簸。”这个就不像是个好标题，因为缺乏概括性；而“飞船有问题”也不是一个好标题，因为太抽象。标题的意义在于，把此缺陷和彼缺陷区分开来，这样，如果曾经研究过，就能回想起大概是怎么回事儿；还在于，让人一眼看去，就能知道大概是应该分配给谁去解决。

还有一些字段能起到类似作用，比如关键字（Key Words）字段。通过关键字，能够知道大概是哪方面或哪些方面出了问题。如果有人对相应方面比较熟，那就考虑把缺陷分配给他解决，或者先向他咨询一下，搞清楚用户遇到的是什么问题，同时可能获得一些关于谁适合去解决这个缺陷的线索。

如果在缺陷跟踪系统里同时收集几个产品的缺陷，那么产品名称字段也就必不可少。类似的，如果单个产品的规模比较大，而又可以划分为几个主要功能模块，那么，模块或者组件字段就会比较有用。

以上字段都是提供简略信息、大致方向，有利于作出初步判断的。而对于真正修改源代码以修复缺陷的开发人员而言，是显然不够的。他们需要对缺陷的详细描述，这通常通过让缺陷报告者填写缺陷的细节描述（Description）区域来实现。

对缺陷的详细描述，一定要做到令开发人员和测试人员可以复现这个缺陷。要详细记录当时的操作环境、当时的操作步骤，以及当时的缺陷现象。这样，开发人员按当时的操作环境搭建自己的操作环境，按当时的操作步骤逐步执行，应该看到与记载相一致的缺陷现象。而当开发人员改好后，再按当时的操作环境，执行当时的操作步骤，缺陷现象应该不再发生了。如果测试人员在重复操作后，也看不到这个缺陷了，那就说明缺陷被修复了。由此可见，对缺陷的详细描述是处理缺陷和检验处理结果的依据，十分重要。

如果按照缺陷的详细描述不能复现这个缺陷，开发人员就无从着手，需要回头问用户，要求用户进一步描述。而如果此时用户也复现不出来这个缺陷，那就比较麻烦了，因为缺陷很可能还在，我们却揪不出来它，不知道啥时候它会再蹦出来。

对如何复现缺陷的详细描述，还应该简洁和针对本质。比如这个描述就不太好：“我敲了大约 5000 字的文档，然后全选，然后按下黑体按钮和斜体按钮，程序就死掉了。该文档是关于我的 2007 年十一长假的旅行见闻，下面我在这里重新输入……[8]”而这个描述就很好：“在一个空白文档上输入特殊符号￥，选中￥，按下黑体按钮，再按下斜体按钮，程序就不再响应。”

在有些缺陷跟踪系统里，除了可以用文字详细描述缺陷外，还可以上传相关附件，比如，当时缺陷情况的截图或使得程序报错的输入数据文件、软件崩溃时的工作日志等。在有些缺陷跟踪系统的详细描述区域不仅可输入简单文本，还可以设置字体、颜色等，甚至直接贴图。这可能是用 Wiki 实现的。

对缺陷所在的程序版本的记录同样重要，这通常是一个单独的字段。这个字段用来记录是使用哪个版本的程序出现的问题。开发人员并不一定真的用那个版本的程序来复现缺陷，而可能是用手头的程序版本复现。如果复现了，就开始修复工作。但是，如果手头的程序版本不能复现，则需要进一步查明原因。

[8] 此处略去 5000 字，其中包括一个￥符号。

是缺陷报告错了，还是手头的版本已经修复了这个缺陷。还是缺陷在手头的版本里隐藏起来了。这时候，就需要尝试在缺陷记录里记录的那个版本上去复现，进而确定不能在当前版本上复现的原因是什么。

9.4 消灭它？立刻消灭它？

我们曾经提到过，新创建的缺陷条目，要经过一道初筛，筛掉假缺陷，筛掉重复的缺陷，剩下的才是“好”缺陷。然而，并非所有的“好”缺陷，都可以消灭，或都可以马上消灭。因为很有可能我们的时间和精力不足以解决全部问题。

这里有一个类比。在灾难性事件发生之时，医疗小组赶到现场。面对众多的伤员，应该做的第一件事情是什么？不是止血缠绷带，而是现场初检（Triage）。初检不同于临床诊断，目的是尽快将被救人员简易分类以便于救护人员识别，并给予不同的处置。伤员被挂上不同颜色的布条，或给带上不同颜色的袖章。颜色分为红、黄、绿、黑四种，分别对应着重伤、中等伤、轻伤和死亡[9]。根据清晰可见的颜色，后续人员进行相应处理：红——立即处理；黄——次优先处理；绿——延期处理；黑——算了吧。这种方法，可比谁叫得欢就抢救谁要合理得多，因为叫得欢的，基本上不是重伤员。

修复缺陷，也同样不能谁叫得欢就先修复他所报告的，也不能让开发人员随便挑，喜欢修复哪个就修哪个。严格按照先后顺序也不对。那凭借什么来排先后呢？救护是依据伤员布条的颜色，修复缺陷也有类似的属性。它们是：严重程度和优先级。

严重程度（Severity）字段，用有限的几个值来表示缺陷对产品使用的影响程度。比如可能是这么定义的：

[9] 这个标记方法并不一定规范，这里仅作为例子。

- 轻微

缺陷影响到了产品功能说明书里没有直接提到的一些次要功能。

- 一般

缺陷影响到了一部分产品功能说明书里直接提到的功能。有时候可以通过一些操作方法来回避缺陷，尽管有些“绕远”。

- 严重

缺陷使得某些重要的功能不能使用。

- 致命

产品不能启动，或者不能执行一些最基本操作。

显然，越严重的缺陷越需要得到关注。严重程度可能先是由缺陷的创建者输入的，不过，他们确定的严重程度应该是可以修改的。产品经理或者研发经理或者测试经理应该对此有最后的发言权。

与严重程度字段并列的是优先级（Priority）字段。优先级字段的值与严重程度字段的值有关，但又不是完全由它确定。比如，如果一个重要客户要求，某个缺陷必须在下个版本中解决，否则不付款，那只好把它的优先级排高了。再如，如果有个缺陷，虽说中度严重，可是解决起来需要改变程序底层架构，伤筋动骨啊，那还是等下个大版本发布的时候再说吧。但是要在即将来临的这个发布版本里，把说明书改改，说明禁止这样的操作。

下面是优先级字段定义的一个例子。

- 低度

描述：是个缺陷，应该解决。

最后期限：缺陷记录创建后的 3 个月之内解决。

- 中度

描述：对产品的成功有一定的影响。

最后期限：下一个主要发布版本中，必须解决。

- 高度

描述：对产品的成功有重要的影响。应该尽快解决。

最后期限：下一个补丁版本中，必须解决。

- 最高

描述：对产品的成功至关重要。应该立即解决。

最后期限：立即解决，增发补丁版本。

先消灭哪个缺陷，后消灭哪个缺陷，由优先级字段的值决定。另外，优先级字段和严重程度字段并不是必需的。对于规模比较大，同时出现的缺陷比较多的开发团队会比较有用。而对于规模比较小的团队，通常就不是很有必要。

尽管我们对缺陷排了优先级，但是仍然要遵循一个原则，那就是，避免带着许多缺陷前进，缺陷应该尽快被消灭。在项目开发过程中，总是有来自客户的压力、来自上级的压力，要求项目经理尽快前进，为此而集中精力于开发新的功能，而不是尽快修补已经发现的缺陷。但是要知道，每个缺陷都需要被记录和管理。积压的缺陷越多，每个缺陷存在的时间越长，就要花费越多的时间对它们进行分类，排优先级，排进度，跟踪，报告。平均到单个缺陷上，其管理成本就会变得很高。另一方面，大量的缺陷会相互扭结在一起，难于识别，难于处理。同时，大量缺陷的存在，会影响到开发人员的新功能开发——基础尚未打好，如何起高楼。

本章至此，我们讨论了一条缺陷记录可以具有的各种属性、各种信息。而对于一个公司、一个开发团队，则应该根据自己的实际情况，对缺陷管理系统中缺陷

记录的字段进行明确的规范，以便对相关信息的理解能够统一，同时也利于分析统计工作的开展。对缺陷相关数据的分析统计工作，是下一小节的内容。

9.5　分析统计缺陷相关数据

如果研发团队很小，比如三四个人，那么领头的人就会对几乎所有发生的事情都很清楚，就会对项目的进展状况、当前状态有很好的把握。但是，如果团队变大，变得很大，那么仅靠一个又一个具体的事情来把握项目的整体情况，就很困难了。因为信息太多了。这时候，信息的分析处理、信息的提炼，通过宏观的数据，帮助管理者把握总体状况，做出相应决策，就变得很重要。这是进行数据统计的主要原因。

在缺陷管理方面也是这样的，一般来说，有三类典型的统计，这三类统计各有益处。

第一类统计用来回答缺陷是怎么分布的、数量是多少。比如，项目目前有多少被发现但还没有解决的缺陷，其中特别严重的有多少，不太严重的有多少。项目经理会据此判断，应该把多少人力投入到缺陷修复上，多少人力投入到新功能上。带着很多缺陷前进可不是件好事。再如，近一个月来，张大侠修复了多少个缺陷，王大侠修复了多少个缺陷。领导会据此判断，谁比较辛苦，谁比较闲。当然，这样的判断很粗糙，因为有的缺陷修复起来很难，有的很容易。但在统计上，这些数据能在一定程度上反映实际情况。这一类统计数据，如果反映在图形上，是分布图。这样的分布图，可以表现为柱状图（见图 9-3）、饼图等。

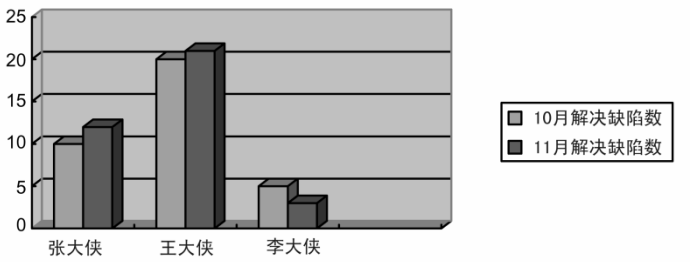


图 9-3　分布图（劳苦功高的王大侠）

第二类统计用来回答，随着时间的迁移，缺陷的总体趋势是什么样的。比如，发现的缺陷的数量，随时间是如何变化的。这个数值一定是随时间不断上升的。但是上升的趋势在项目末期应该减缓，因为产品越来越稳定，越来越难于发现新的缺陷。如果画图表示，以发现的缺陷数量为纵轴，以时间为横轴，这条曲线应该是始终上升的，但是越来越缓。再如，已经修复的缺陷的数量，在任意时刻应该始终比发现的缺陷的数量少。并且随着时间的迁移，越来越接近发现的缺陷数量。两条曲线对比，说明了尚未解决的缺陷的数量随时间变化的趋势。这些对于项目经理把握项目情况会有所帮助。如果新发现的缺陷越来越少，而绝大部分缺陷已经被修复，大概就可以发布产品了。这一类统计数据如果反映在图形上，是趋势图（见图 9-4）。

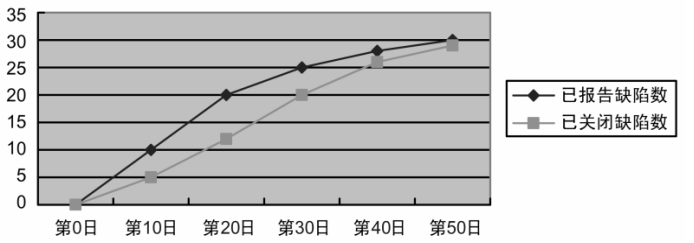


图 9-4　趋势图

第三类统计，是关于缺陷的年龄的统计。准确地说，是缺陷处于某一特定状态的持续时间的统计。比如，已经提出一周但是没人解决的缺陷有多少，已

经提出三周但是没人解决的缺陷有多少，已经提出半年却还没人解决的缺陷有多少，具体是哪些。这类统计特别有利于发现那些被遗忘的缺陷，也有利于发现那些特别棘手的缺陷。这一类统计数据如果反映在图形上，是时长图（见图 9-5）。

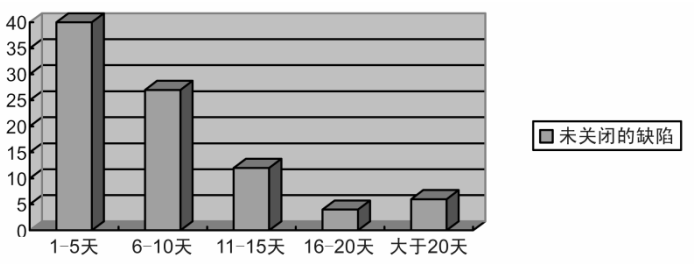


图 9-5　时长图

除以上三类统计外，还有一些用于缺陷分析的统计。比如缺陷密度（Defects Density）——平均每一千行代码修改引入的缺陷的数量；比如缺陷逃逸比率（Fault Slip Through）——有多少缺陷在产生它的阶段没有发现；比如缺陷触发分类（Fault Trigger Classification）——缺陷是通过什么类型的测试暴露出来的。这类统计，有利于改进软件开发流程，调优质量保证工作的方法、时机和程度。

图形的方式是展现缺陷相关统计数据的非常好的方法。作为另一种展现方式，报表也不错。报表的好处是，文字上的东西可以多些；具体的数字，可以看得更清楚些。更广义地说，查询也可以认为是一种数据展现方式：根据特定的查询条件，得出查询结果，通常是一个列表。这对于跟进到具体问题，是很有好处的。

这些数据的采集和展现，应该尽可能得到缺陷跟踪工具的支持。人工来做，耗时费力。

> **[Bugzilla 等开源工具]** 通常查询是灵活的、可以定制的。大多能够提供一些常用的报表。对图形表达的支持可能欠佳。

> **[ClearQuest]** ClearQuest 能够很好地支持相关数据的分析统计，并且可以以图形、报表、查询等多种方式表达出来。图形、报表和查询都是灵活的、可以定制的。

9.6 关联缺陷记录与变更集

还记得变更集么？变更集是指一个明确的工作任务所对应的源代码改动，带有明确的目的，可以和其他工作任务区分开。修复一个缺陷，通常就是这样的一个明确的工作任务。因此，通常可以把一条缺陷记录和一个变更集关联起来。

这样做的主要好处是，能够比较直观地显示，这个缺陷的修复究竟改动了哪些源文件，改动了哪几行源代码，便于日后查看。比如下面这种情况，就需要这样的查看：张大侠修复了一个缺陷，提交了相关的变更集。在系统集成后，测试团队进行测试，看看这个缺陷是不是真的修复了。结果发现，只有在某些情况下，这个缺陷的现象才不会出现了。而在另一些情况下，这个缺陷的现象仍然会出现。于是，测试团队更改了这条缺陷的状态，通知张大侠。张大侠根据这个缺陷条目，找到相关的变更集，进而分析改动的源代码，看看为什么这个缺陷有时候还会出现。

这是一个方向的查找，由缺陷记录找到变更集，进而查看相关的源代码改动。有时候，需要反方向查找，根据变更集查找缺陷记录，进而查看缺陷的具体情况，比如缺陷现象等。因此，关联缺陷记录与变更集要支持两个方向的查找。

如何实现关联？一个简单的方法是，在变更集的标题或描述中，记录相应缺陷条目的编号。这样可以方便地根据变更集查找到缺陷条目，反之也一样。

有些工具提供对这种关联的内在的支持，这样查找起来更方便，只需要按个按钮，甚至连按钮都不用按，就已经把相关内容显示出来了。对这种关联的内在的支持，其好处还体现在变更集创建、缺陷记录状态改变等方面的辅助和自动化。当决定修复一个缺陷时，按个按钮就自动创建了相应的变更集，并关联起来。当相应的变更集完成并提交时，自动地改变缺陷记录状态，标志这个

缺陷已经修复完毕。

> **[Bugzilla 等开源工具]** 通常不提供这种关联的内在支持，但可以考虑通过对工具的二次开发来实现。
>
> **[ClearQuest]** ClearQuest 与 ClearCase UCM 配合，提供了这种关联的内在支持。

需要提及的是，缺陷记录与变更集之间的关系，并不一定总是一对一的关系。有的缺陷的修复需要对程序有比较大的调整，需要多个人一起配合，于是需要有多个变更集。而另一些时候，几个相关的缺陷可能通过一个变更集就都修复了。

在当代版本控制工具里，每提交一个变更集就会生成一个新的整体版本。这个新的版本有一个版本控制工具分配的自然版本号，还可能有一个有意义的版本名称。因此，变更集和代码整体版本常常都是可以用版本号表达的。我们在此做一个小结，看看变更集/版本号跟缺陷记录在哪些地方产生关联：

- 在哪个版本发现了该缺陷？
- 该缺陷是通过哪个/哪些变更集修复的？
- 测试人员在哪个版本上做的测试，证明它已经被修复了？

以上的讨论，实质上是关于源代码版本控制功能与缺陷跟踪功能的集成，还有更多的工具、更多的功能值得相互间交互和集成。比如，不少缺陷跟踪工具也内嵌了一些项目管理的功能。而缺陷跟踪工具中的一个条目，也值得考虑跟测试管理工具中的一条或多条测试用例相关联。还有需求管理工具、建模工具、构建管理工具……把它们集成到一起，作为一揽子解决方案，被称做**应用程序生命周期管理**（Application Lifecycle Management，ALM）工具集。在介绍源代码版本控制工具时（第 2.3 节）提到的 TFS（Team Foundation Server）和 RTC（Rational Team Concert）是它们的典型代表。

第 10 章　管理变更　10

比缺陷管理更广阔的话题，是变更请求管理。比变更请求管理更广阔的话题，是问题管理[1]。

[1] 雨果说过类似的话。雨果，Victor Hugo，1802—1885 年，法国大文豪，是法国浪漫主义文学运动的领袖。

10.1　管理增强

第 9 章所讲的跟踪缺陷的方法，也同样可以用来管理其他一些事情。填报一个缺陷条目，本质上是提出一个要求对程序进行修改变更的请求，请求修复这个缺陷。而请求修复缺陷只是**变更请求**（Change Request，CR）中的一种。变更请求还可能是要求对程序源代码进行修改，以便对程序功能进行细微的增强或显著的提升。不仅是源代码，对任何已有内容、已确定的计划进行改变的请求，比如变更需求、变更设计等，本质上都是变更请求。对变更请求的管理是本章的主要内容。

变更请求的基本特点是条目化，需要用类似于管理缺陷那样的管理方法来管理：记录并跟踪每一个条目，直到该条目完成或关闭。不仅变更请求是条目化的，还有很多事情是条目化的，需要按条目记录和跟踪，并试图解决。比如客服人员记录客户提出的各种待解决问题，项目经理记录项目中所有悬而未决的事项，等等。我们把这些统称为**问题**（Issue），相应的，使用**问题跟踪工具**或称问题跟踪系统（Issue Tracking System）来管理它们。

以上是对变更请求和问题跟踪的概念的介绍。下面我们是要详细介绍变更请求中的一种，它的名字叫增强。

增强（Enhancement），是指一些小小的请求，请求把程序已有的功能，稍稍增强一点，或者稍稍改变一下。比如，可能是用户操作到这儿了，觉得这儿该有个操作提示，可实际上没有；可能是用户希望字体再大一点，显眼一点；可能是用户觉得备选的 20 种表情图标不够用，要求再添上一种。总之，应该是对功能的琐碎的要求，提出进一步改进的期望。

对这类请求的管理，挺像对缺陷的管理。缺陷的特点是，数量众多容易丢；通常每个涉及的改动工作量都不大，经常是一个变更集就能完成；通常每个缺陷能明确对应到源代码的改动；通常缺陷之间相互独立，不需通盘考虑。正是

因为这样的特点，所以把缺陷按条目记录，进而分别跟踪、处理，直至解决。

而增强是基于已有产品功能之上的小的要求、小的修改。这样的要求可能会有不少，容易弄丢，因此需要跟踪；这样的要求通常容易实现，费不了多少时间，也不会对程序造成伤筋动骨的影响，也不会对进度有太大的影响——只是表面打打蜡，添朵花儿。由于不会伤筋动骨，所以各个功能增强之间的关联就不大，通常可以单个考虑，逐一实现。这就是增强的特点，很像缺陷，所以管理起来也很像是管理缺陷。对功能增强同样要报告，以一条记录的形式存储在数据库里。之后，它的状态不断改变，直到最后确认已经实现。

但是，对增强的处理流程与对缺陷的处理流程，也有所不同。这主要体现在两点上。首先，需要对增强有更仔细的评估，进而确定是否要实现，以及实现的优先级。在缺陷管理里，也要进行技术上的初检和评估，确认它是个“好”缺陷。如果它是个“好”缺陷，那么通常它就应该被解决。而增强则不同，很多增强请求是不会被实现的。这里要考虑的因素比较多，既要考虑市场对它有多需要，也要考虑实现它有多大的代价和风险，考虑时间是否允许，人力是否允许。

其次，增强所导致的修改，常常不仅是源代码的修改。如果有详尽的需求文档，那么需求文档有可能要因此修改；用户使用手册通常也需要因此而修改；测试用例、测试脚本，也有可能需要因此而修改。在每个增强条目中，如果对这些修改有所提醒或有所记录，则比较好。

10.2 管理特性

增强是一些细小琐碎的需求。如果需求不是那么细小琐碎呢？在一些软件开发流程中，仍然用类似于管理增强的方式来管理它们：记录并跟踪每一个条目，直到该条目完成或关闭。这时，这些条目常被称做**特性**（Feature）。

典型的，**Scrum** 使用类似的方法。Scrum 是一种以迭代、增量为主要特点的软件开发过程，是敏捷软件开发方法的一种。在 Scrum 中，所有的待完成的特性都列在产品订单（Product Backlog）里。每次迭代前，从这里挑出来本次迭代要实现产品订单中的哪些条目，把这些条目列在冲刺订单（Sprint Backlog）里。为啥叫冲刺订单呢？在 Scrum 里，迭代被称做冲刺（Sprint）。每次迭代通常是 15 到 30 天。在迭代的过程中，一般不会改变冲刺订单。在一次迭代完成后，回顾工作成果和经验，再计划下一次迭代。

在上一节我们讲解管理增强时，所有的增强请求放进了变更请求管理工具，或称问题跟踪工具。而在 Scrum 里，所有的待完成的特性放进了产品订单。管理增强时，我们为每条增强请求确定优先级。而在 Scrum 里，我们把本次迭代要完成的特性放进冲刺订单。管理增强时，每个增强请求经过一系列的状态转换，最终关闭。在 Scrum 里，在一次迭代完成后，逐条检查冲刺订单中的特性，确认已经完成。看，尽管具体手段不同，但它们本质上很像：按条目管理需求，并按同样的条目为管理单位，管理开发，实现功能，集成在基线里，最终对外发布。

不仅是 Scrum 用这样的方法来管理需求，另一个典型的例子是**特性驱动开发**（Feature Driven Development，FDD）。在该方法里，通过一系列的方法，需求被分解为各个特性。特性都不大，但都意味着对用户是有价值的，程序外在的功能。每个主开发人员（Chief Programmer）每次选择几个特性，在未来的两个星期里带领他的同事们一起完成。完成后，提交到主干。在该方法里，条目化的需求也同样是开发管理的主要线索：识别特性、建立特性列表，按特性制定计划并跟进，按特性设计和实现，按特性集成进主干。总之，按条目化的需求组织开发工作。

缺陷、增强、特性，从配置管理的视角看，它们都是变更请求，请求对程序的外部表现进行变更。每个条目会落实成具体的源代码改变。这些代码改变

也常被称为变更，体现为变更集。作为源代码改变的结果，在集成产生的版本中，这些变更被实现了，变成了用户看得见摸得着的程序版本之间的功能变化。这种变化，也被称为变更。从这里可以看出，变更在不同的语境里，可以有不同的含义。

变更的含义还不止这些，在下一节里，我们回到古老的瀑布模型看看。

10.3 瀑布模型中的变更

瀑布模型希望能够借助在传统工程项目中的成功经验，来实现软件开发项目的成功。传统工程项目，比如盖房子，大体上是这样的：先弄清楚需求，要在哪儿盖，盖个啥东西，大概预算多少，等等。然后给出大致的设计，大致的模样。进而给出很详细的设计，包括空调管道的设计、电力系统的设计，等等。这些都确定了，就开始把盖房子的任务拆成很多个小任务，定时间、定人员，然后按部就班地做，直到房子盖好。

瀑布模型希望也能用这样的方法进行软件开发。首先做需求，弄清楚客户想要什么，自己想做什么，把这些写成正式文档。这要和包括客户在内的与此相关的人员反复讨论、评审，直到所有相关人员达成一致意见，于是需求就确定下来了。而需求文档在这个时候的版本，被称为需求的基线[2]。在需求确定下来之后，再做设计。如果是大项目，可能再细分为概要设计和详细设计。设计的成果，也就是设计文档，也要反复讨论、评审，直到所有相关人员达成一致意见。于是设计就确定下来了。或者说，形成了设计的基线。设计完成后再做实现，也就是编写程序代码，并且集成在一起，测试通过。实现完成后，发布产品给客户。

请注意，这里面的每一个阶段工作的启动，都是基于前一个阶段工作所完

[2] 是的，基线在这里的含义，与谈论源代码整体版本时谈到的基线的含义是有区别的。但它们都体现了基线的本意：基准和参考。

成的，是明确的、确定的工作成果，也就是需求基线、设计基线等。瀑布模型希望这些基线的产生是非常谨慎的、深谋远虑的，以至于一旦确定，就不再变化。这样固定下来，才能作为下一阶段工作的指导。然而，现实是残酷的，怎么可能一旦确定就不再变化？于是，作为变通，基线可以变化。然而，这样的变化必须是非常谨慎的，并且是尽可能少的。否则，使用基线的人就会犯晕，因为变来变去，搞不清楚到底应该做什么；或者犯急，因为在预定的时间内没法完成这么多追加的事情。

因此，在瀑布模型中，变更的管理很重要。其主要的关注点是，一旦定下来的事，就要执行，不能乱改。要减少变更，变更不能随意发生。当然，也不能禁止变更的发生，因为有时候变更是不得已的。一根筋地往前走，见了车都不躲一躲，这不像是个理智的行为。所以，变更管理的核心是，严格控制变更。

要充分评估，充分分析变更的影响，分析做这个变更与不做这个变更相比，会有什么样的收益，会有什么样的代价，会有什么样的风险。这可能涉及项目资源、时间和任务的分配；涉及系统架构改变；涉及用户体验、客户关系，等等。评估的过程中，可能要征求各方面意见。在评估之后，要进行决策。决定这个变更，要不要接受并实施。这也同样需要征求各方面意见。

一个常见的做法是，设立**变更控制委员会**（Change Control Board，CCB），包含所有各方面的代表。大家坐在一起，对变更进行各方面的评估，并对评估结果达成一致；大家对是否接受这个变更进行讨论，并最终形成决策。

而一旦做出了实施的决策，就要告知所有相关的人。当修改了需求文档、设计文档，就要保存这个新的版本，并且让大家知道有新版本可用。还要考虑继续跟踪，保证这个变更最终在产品中实现，正确地实现。

尽管添上了变更控制环节，以弥补瀑布模型过于理想化的倾向，瀑布模型仍然存在问题。因为软件开发在本质上与盖房子这样的事情不一样。事实上，

设计工作常常会引起需求变更，因为在设计的过程中可能会发现需求没法实现或成本太高。代码实现工作常常会引起设计变更，因为在代码实现的过程中可能会发现设计有重大缺陷，无法实现。更为严重的问题是，用户常常是在看到并且试用软件产品的时候，才能清晰地表述他到底想要什么，这可能和项目初始时，空对空确定的需求很不一样。而当问题在这个阶段暴露出来的时候，常常导致项目的延期或成本的大幅上升。

而变更控制的流程本身也耗费很大。CCB 好像应该把所有的角色所有的人都包括进来，因为指不定什么时候就会跟自己或哪个人相关。CCB 的会议好像永远也开不完。

上述这些问题，使得软件开发团队越来越倾向于迭代的、增量的开发方式，让每次迭代需要开发的内容较少、容易确定；在设计和实现时，及早发现问题；在功能实现后，及早获得用户的反馈。项目早期的迭代，会把比较多的注意力放在典型的、最重要的需求上，会把注意力放在产品的整体架构上，然后，通过编写代码以验证关键设计，通过向用户展示可运行的程序以验证关键需求，尽早得到这些关键问题的反馈。这样，以后的风险和难度就小多了。

上述这些问题也使得软件开发团队越来越倾向于按需求条目来组织设计和实现。这样，需求问题、设计问题、代码实现问题的沟通和讨论都会迅速缩小范围到相应的特性开发团队中，这比瀑布模型中全局性的需求收集及其变更、设计及其变更、开发协调管理工作要有效率得多。

当然，按需求条目组织设计和实现，也不能教条。有时候，几个需求适合作为一个开发任务实现。有的时候，实现程序的框架相关的开发任务并不适合于对应到某一需求。还有的时候，需要把重构作为开发任务。不妨每次迭代的任务列表中，尽量以需求条目为主，但也允许其他类型的开发任务。

10.4　影响变更管理的因素

不论是对缺陷、增强和特性的管理，还是对瀑布模型中变更的管理，本质上都是条目化的管理：描述条目并请求实现；审核该条目并排优先级；修改源代码或文档以实现该条目；验证和关闭该条目。

作为严格的变更管理，需要在变更提出时填写足够详尽的相关信息。然后是足够的变更评估，评估变更影响的方方面面。这期间，要与足够广泛的相关人员交流。之后，是做出变更决定。这也要经过充分讨论，并最终取得一致。这个决定，要通知到所有可能相关的人员。接着，是变更的实施，对源代码和文档的具体改动。要把变更引起的具体的源代码改动、文档改动与该变更记录联系起来，记录下这种联系，以便将来查找核对。变更实施后，要经过足够的测试和验证，保证变更真的发生了，真的正确地发生了。这之后，要把变更确实发生这件事，再告诉所有可能相关的人员，让内部人员记得取文档的最新版本。让客户知道，本版软件与上一版相比，发生了什么变化。

显然，不是所有的变更，都要经历这么严格的管理；不是所有的变更，都要经历相同程度的、相同模式的变更控制。比如说，如果张大侠在 5 秒钟前想给某个函数内部的变量起名叫 nSize，5 秒钟后改了主意叫 nCount，那么他根本不需要为此填一条变更请求，并找领导批准。

那么，哪些因素会影响对变更的控制程度和控制流程呢？

- 变更的类型

缺陷一般来说都会被修复，而功能增强、新功能增加这类的变更则不同。即便是正确有效的，也很可能因时间、成本等因素不去实施。对它们的分析评估变得很重要。

- 变更的大小

有些变更可以很快完成，而有些变更可能需要不少的人力和时间，会影响到整个项目的进度。对于大型变更，这意味着一个重大决定，当然需要更严格的控制和更高层的介入。小的变更，可能测试人员填一条记录，指定开发人员，然后这个开发人员直接就改了；大的变更，就需要项目经理来决断；更大的变更，要更谨慎，可能是项目经理、系统架构师、用户代表等人，一起仔细讨论，才能决定。

- 变更的影响面

有些变更，只会影响产品的局部；而有些变更，则可能会产生广泛的影响。有些变更，只对打算变更的人自己有影响；而有些变更，则会影响很多人。比如，如果一个变更请求涉及某个公共函数的少量修改，而整个产品的上百个地方都在调用这个公共函数，那么，尽管修改本身的工作量不大，但是它的影响面却很大。这时候，就应该谨慎行事，并且和尽量多的人沟通。在变更前，应该广泛征求可能受影响的模块负责人的意见（这可能是在某次会议上完成的），在变更后，应该告知大家，发生了这样的改变，请进行相应检查（这可能是通过一封电子邮件完成的）。

- 变更的风险

当变更的风险比较大的时候，需要谨慎行事。那么，什么时候变更的风险比较大呢？一方面，如果变更的影响面比较大，一旦出了问题造成的损失比较大，那么常常风险也比较大；另一方面，风险还受“失败的可能性”的影响——如果失败的可能性、出差错的可能性比较大，那么，即使变更的影响面不大，它的风险也比较大。

- 变更发生时间

变更请求，通常越早提出越好。就好像盖大楼，在做设计的时候提出变更，只需要改改图纸。而在工程完工后提出变更，却要推倒重盖。因此，越到项目

后期，对变更的控制越严格，越不容易接受变更。在项目末期，通常只会接受缺陷修复和小的功能增强，甚至可能只接受重要的缺陷修复。

- 研发规模

在小型团队里，看似随意的沟通常常十分高效。但是，随着人数增加，成本会越来越高，并且会出现信息丢失、相互矛盾等一系列问题（见图 10-1）。

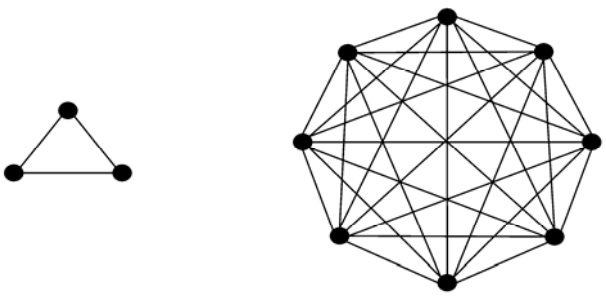


图 10-1　沟通成本可能随人数的增加而激增

这时，必须引入一些固定的、有些形式化的方式，来处理沟通和协作问题。对于变更的管理也是一样。比如建立变更控制委员会，比如变更请求条目的状态发生改变时，自动发邮件。当然，如果能够把大型团队分解成相对独立的小团队，那就最好了。按特性组织开发工作就是这个思路，我们已经提到过。

10.5　控制产品版本间的差异

要控制产品版本间的差异。该完成的，要完成；没必要做的，不要做。需求分析人员应该对此负责，尽力搞清楚用户到底想在下个版本里要什么。系统设计人员应该对此负责，尽力让设计的实现能够满足本次版本的需要，同时避免无用的设计。项目经理应该对此负责，做好任务分配、项目进度监控，让应该完成的都按期完成，而不需要做的不去做。听起来这些都跟软件配置管理关系不大，下面我们谈谈有关系的。

在版本控制工具里，可以考虑加一些控制，以保证开发人员做且只做该做的事。一种方法是，让变更集不由开发人员创建，而是由管理人员创建，然后指定给相应的开发人员完成。或者，要求变更集必须有对应的变更请求，而变更请求是管理人员批准的。这样做是为了控制开发人员要做的事。对于一些比较大的任务，需要不止一个人，不止一个变更集完成的任务，可以考虑为此开分支。分支由管理人员创建，然后要求相应的开发人员在上面工作，完成相应任务。如果在特定的版本控制工具中，不方便实现这样的控制，那么可以考虑制订类似的流程，要求大家遵守。不论是否由工具支持和强制执行，这些都是在修改代码之前，事先控制的方法。

另一类方法是修改代码之后提交时控制。在开发人员完成了变更集后，如果想提交，需要得到某种批准，只有符合本次版本需要的变更集才可以提交。分支也是这样，只有符合本次版本需要的分支才允许合并到主干。

还可以再往后拖一些时间再控制。可以在（狭义）集成开始前，或集成后送交测试前，检查本次版本所包含的新的变更集，检查这些变更集的名称或说明。一方面，这些变更集必须都是为实现本版本目标服务的；另一方面，本版本要实现的变化，都有相应的变更集对应，都已经由它们完成了。如果除了变更集外还使用了分支，那么也要做类似的检查。一方面，这些分支必须都是为实现本版本目标服务的；另一方面，本版本要实现的变化都有相应的分支或变更集对应，由它们完成。

要注意的是，这些都是非常严格的控制，在大多数情况下，没有必要进行这样控制，因为这样的控制是有管理上的成本的。但是在有些情况下，比如，在临近项目结束，产品即将发布的时候，可以有选择地采用上述方法，以保证集中精力在最重要的事情上，并且尽可能少地引入风险：新增功能和不重要的缺陷修复不仅会分散精力，而且可能会引入新的缺陷。

10.6　表达产品版本间的差异

用户想知道不同发布版本间的差异。用户想从中知道新增了哪些功能，哪些功能的使用方式发生了变化，哪些缺陷在这个发布版本里被修复了。

测试人员想知道不同送测版本间的差异。测试人员想知道从上次测试过的基线到这次待测试的基线，修复了哪些缺陷，新增了哪些功能，哪些功能有变动。甚至测试人员还可能关心，源代码的哪些部分发生了改动，要重点测试。

开发人员想知道自上次更新工作区以来，主干上又有了哪些新的提交，引起了程序外在功能的哪些变化，这些变化是哪些源代码改动引起的。对于与自己所在模块相关的改动，开发人员会格外关心。

因此，需要以一定的方式表达产品版本间的差异。有四个途径可以获得这类信息，以整理汇总，表达版本间差异。

第一个途径是查看版本控制工具。版本控制工具中记录了每个变更集所完成任务的简述。查阅在两次发布版本间提交的所有变更集，就能知道做了哪些工作，也就能推导出，两个发布版本间外在的功能差异。如果两个发布版本间距离比较近，变更集不多，这个方法会比较好。如果其间的变更集很多，恐怕就不太有效。

第二个途径是查看问题跟踪工具中的变更请求。这对获得与缺陷相关的信息特别有效，查询哪些缺陷在新旧两个版本间修复过，把这些缺陷的名称和简述列出来就可以了。用类似的方法对获得与增强相关的信息也比较有效。

第三个途径是查看项目计划或迭代计划。这些文档，可能给出了每次发布的主要的开发任务。可以据此导出，这次发布的程序与上次发布的程序之间主要的不同点。

第四个途径是查看需求数据库或需求文档。如果需求数据库或需求文档中，

对每一条需求，明确标出了在哪个发布版本中首次实现，则接下来的工作会很好做：只需要把此发布版本对应的需求整理出来，就可以了。

在实践中，以上四个方法可能需要综合运用。

用户所关注的版本差异，主要是产品外在功能、性能方面的变化，特别是过去版本中缺陷的修复，和当前版本中的重要功能增加和改变。用户关心这些，以决定是否要购买，购买哪个版本，或是否要从旧版本升级到新版本。这些信息通常写在**发布说明**（Release Notes）文档中，提供给用户。撰写发布说明，通常需要根据以上四个途径得到输入信息，进而整理和概括。

而相比之下，测试人员和开发人员，尤其是开发人员，还会关注程序内部的变化，关注源代码改动本身。同时他们关心的粒度也更细：最近具体哪些开发任务完成了，这些内容主要是通过第一种和第二种途径得到，而且不太需要进行整理和概括。因此，可以考虑使用自动化的方法得到这样的差异。比如：

- 每次集成后，由集成工程师运行脚本，产生完成条目的列表。
- 在持续集成工具中设定，每次集成后，产生完成条目的列表。
- 开发人员和测试人员直接访问版本控制工具和问题跟踪工具，由工具产生完成条目的列表。

注意，当使用持续集成策略时，产生基线的数量会很多。测试人员测试的两条基线之间，开发人员相邻两次更新工作区所用的基线之间，可能存在着大量的基线。因此，集成时为每个基线产生一个完成条目列表的方法，可能会变得不适用：使用者会觉得翻看这么多列表很麻烦，特别是当这些列表是 Excel 表或 Word 文档的时候。可以考虑的一个改进方法是，通过一个网页界面，选择任意两个基线，就可以自动列出其间所有的变更集，以及完成的缺陷修复、增强和特性等变更请求条目。

第 11 章　玄妙的学院派 11

“配置管理由配置识别、配置控制、配置状态报告、配置审计四部分组成。”学院派这么说。这听起来很玄妙，是不是？在这一章，我们努力对这套传统的理论进行解读，尽量和实际工作联系上。

11.1 配置识别

配置识别（Configuration Identification，CI），又名配置标识，其定义如下：

“An element of configuration management consisting of selecting the configuration items for a product, assigning unique identifiers to them, and recording their functional and physical characteristics in technical documentation.”[1]

“它是配置管理的一个组成部分，包括：选择产品的配置项，为它们指定唯一的标识，并在技术文档中记录其功能和物理的特征。”

如果要造汽车，总得弄清楚，需要哪些零件，各是什么型号。这里，零件就是产品的配置项，零件名称、型号就是它们的唯一标识。零件总得有零件设计图和使用说明之类，这些分别是对零件的物理特性和功能特性的描述。

看，这个定义套在对硬件的配置管理上，很合适。考虑到软件配置管理理论起源于硬件配置管理，这也就不足为奇了。那么，如何对软件开发实施这样的管理呢？可以从两个角度理解：从人做管理的角度或用自动工具的角度。

当你打算开始软件配置管理工作的时候，你总得先弄清楚，要管哪些东西。我们已经知道，软件配置管理，管的是软件资产：不仅要管源代码、要管文档、要管安装包，还要管环境和工具，比如编译工具等。这是一种划分的方法。如果换一种划分方法，我们还可以说，不光要管交付给客户的东西，也就是安装包等，还要管研发过程中产生的东西，比如源代码，比如分析设计文档，比如测试用例等。此外，还有些东西是从外界取得的，比如，可能产品源代码的一部分是从网上抄来的，或者说是使用了开源项目的成果。

这么多的东西要管，总得一样一样地标出来。这并不意味着一定得一个文件一个文件地标，那太累了。源代码通常作为整体标出来就可以了。也有的时

[1] 出自 IEEE-STD-610，并被 CMM、CMMI 引用。下文 11.2、11.3、11.4 节的起始，也是这样。

候，出于不同的原因，会把源代码划分为适当的组件或模块，分别标出来。比如有些组件，可能不仅是这个产品的一部分，还是其他产品的组成部分，所以要单独管理。而文档，特别是重要文档，比如需求文档，总体设计文档，可能会管得细一些，要按文件标出来。这样，将来如果它没入库也好查。所以，怎么划分全看如何让工作方便。在术语上，我们管这样标识出来的每一项的特定版本叫**配置项**（Configuration Item，CI）[2]。

显然，每一个配置项应该有个名称或编号，这样就能一眼看出它大概是什么，也就不会彼此重复。这被称为唯一标识。

但是，这常常是不够的，每一个配置项，可能需要更多的信息。比如，它到底是什么，能不能描述得详细一点？它属于哪一类，文档还是源代码？它是我们自己开发的，还是从外面找的？它属于哪个产品或哪些产品？它存储在哪儿，哪个工具里，哪个目录下？这些方面，也需要有所记录。这称为记录它的功能和物理的特性，需要被记录到技术文档里。要注意，定义里所说的技术文档是一个泛指，并不一定是一个文档文件。技术文档可以在软件配置管理计划文档里，把要管的东西列出来、理清楚，进而以此为线索，考虑如何管理，也可以是其他的方法。

上面是从人做管理的角度来理解配置识别，而如果从版本控制工具的角度看，则是另一番风景。在硬件领域，零件组成机器，零件的版本组成机器的版本。机器特定版本的配置，就是各个零件的特定版本。同样，在软件领域，众多源代码文件组成了源代码整体，众多源代码文件的特定版本组成了源代码整体的特定版本。特定源代码整体版本的配置，就是各个源代码文件的特定版本。工具管的就是这个，所以，从工具的角度看，配置项就是各个文件的特定版本。

每个文件不仅有外部的文件名和路径，可以作为外部的标识，在工具内部，通常还有唯一的识别标记，对应于每个文件。这样，即使这个文件改了名字，

[2] 也有的时候不太严格地把每一项而非每一项的特定版本称做配置项。

工具还是能够找得着它，挖出它所有的历史记录。所以，文件作为配置项被唯一标识了。

至于配置项的功能特性和物理特性，对于源代码文件来说，就是源代码文件的内容了。在硬件领域，造表格、写文档的时候，不能把一个真实的零件放进去，所以要刻画它，描述它，把这些内容放进文档，并在日后依据这些内容来生产出这个零件，或者说复现出这个零件。而软件不同，软件就是 0101，软件就是信息本身。那么，把信息保存好就是了。自动工具起到了这样的作用，保存了源文件的版本，因此，它实现了“在技术文档中记录其功能和物理的特征”。

而如果配置项是文档的话，那最好还带上说明信息，详见 8.2 节。

11.2 配置控制

配置控制（Configuration Control）的定义如下：

“An element of configuration management consisting of the evaluation, coordination, approval or disapproval, and implementation of changes to configuration items after formal establishment of their configuration identification.”

“它是配置管理的一个组成部分，包含评估、协调、批准/拒绝、实施对配置项的变更。这应该发生在正式的配置识别之后。”

这里是在讲对变更的管理，特别是对变更请求的管理。变更请求被提出来之后，先要评估它，看看它是什么类型的变更，是缺陷修复，是增强请求，是新特性，还是瀑布模型中对需求或设计文档的变更？看它是不是别人已经提出过的？如果是缺陷，看它是否能复现？如果是增强、新特性及其改变，则更要仔细评估。看看如何才能实现它，给出初步方案；看看它涉及哪些配置项；看看它需要多少人力和时间来实现；看看实现它对项目进度和成本等方面的影响；看看有什么风险没有；看看客户究竟有多么需要它。预测一下实现它或不实现它，客户会有什么样的反映，等等。

对于变更，还有协调的工作要做，这主要体现在三个方面。一方面，要考虑不同的变更请求之间的关联性。有些变更请求，可能说的是一回事儿，或者是相关的事情。如果出现此问题，则需要统筹考虑它们对设计的影响，它们的实现成本；要考虑是把它们一起实现，还是都拒绝或拒绝一部分。另一方面，一个变更请求可能会影响到不同的产品，因为它所进行的改动是在一个公共组件上，而这个组件被好几个产品使用。因此，要统筹考虑好几个产品，考虑要不要批准这个变更，还是拒绝。第三方面，协调也体现在完成的先后顺序上，有的请求比较紧急，有的请求比较重大，要优先安排人力资源去完成，有些则可以拖一拖。

在仔细考查了变更请求的成本、风险、价值等因素后，作出批准或拒绝的决定，以及何时实施、谁来实施等（如果变更获得了批准的话）。这些决定可能是一个合适的人作出的，比如项目经理；也有可能是一伙人作出的，比如变更控制委员会。

在作出了批准变更的正式决定之后，就要实施它。改动相关的文档，改动具体的源代码文件，直到变更请求被实现，被正确地实现。在实现了变更之后，也就产生了正式的新版本，可能是某个文档的新版本，也可能是产品的新版本。

在这里，值得特别强调的是，不同的变更请求，应该实施不同程度的控制，走不同的变更控制过程。比如，对于瀑布模型，就要严些，而对于迭代模型，就会宽些。对于项目初期，可以松些，对于项目末期，可以严些。具体讨论见 10.4 节。

11.3　配置状态报告

配置状态报告（Configuration Status Report）的定义如下：

“An element of configuration management consisting of the recording and

reporting of information needed to manage a configuration effectively. This information includes a listing of the approved configuration identification, the status of proposed changes to the configuration, and the implementation status of approved changes.”

“它是配置管理的一个组成部分，记录和报告用来有效管理配置所需的必要信息。这些信息包括一个已批准的配置识别清单，变更请求当前的处理状态，以及已批准的变更的实现情况。”

做配置管理，需要数据和信息的支持，这些数据和信息，不是配置项本身，而是关于配置项的元数据。这包括三类典型的信息，我们仍以车辆设计为例。

首先，车辆是由哪些零件，什么型号的零件组成的。这应该是一个列表，详细列出零件的名称、编号、型号等。这个列表，应该是经过试验的，确实能组装成车的，应该是经过批准的、正式的一个列表。

第二，车辆设计在正式批量生产前，可能有改动的请求，有的想把车灯弄亮点，有的想开个天窗。都觉得这样将来车才好卖。这些请求一定要经过控制，不能谁想怎么改就怎么改。而控制的方法，就是让这条变更请求，去走 9.2 节里那个状态转换图。变更请求有了状态，我们就能知道下一步该对它做些什么。并且，也便于查找。因此，变更请求的状态应该被记录。

第三，变更实现的情况。比如，在变更得到批准后，仍处于状态转换图中，说明工程师是不是已经开始为此工作了，是否完成了，是否交活了，测试是否通过了，这些改动是否已经在车辆设计的最新版本里体现出来了。变更实现的情况，同样被很多人关心，比如提出者、负责工作分配的经理、测试验收人员，等等。所以，同样要做记录，以利于管理。

而软件开发的配置管理所收集的信息、所统计的信息、可以报告的信息，远比硬件设计要多，也远比硬件设计要丰富。

每个版本库相关的信息可以有版本库的名字、版本库的存储位置、版本库的编号、版本库的管理者、版本库的主要用途描述等。版本库中的每个产品相关的信息可以有产品的名字、产品存储在哪个版本库、在版本库中的哪个位置、产品的管理者、产品的大致描述等。如果一个产品包含了多个模块或组件，如果一个产品的源代码存储跨越了多个版本库，则相关的信息会更多。

每个文件相关的信息可以有它存储在哪个版本库，它属于哪个产品，它存储于哪个路径下等。文件的每个版本相关的信息可以有它的版本号、它对应于哪次签入、它是否被锁定等。文件的每次签出签入相关的信息可以有签出的版本号、签出的人、签出的时间、相关的变更请求的编号、变更的描述、签入的版本号、签入的人、签入的时间等。

每个变更请求、每个基线、每次发布、每次版本库的备份等，同样有丰富的数据和信息的记录。你可能会担心，这些是否意味着繁重的工作，是否意味着繁琐的过程。不用担心，软件的配置管理有着强大的自动工具支撑。这些信息的绝大部分都是被工具自动记录的。

这些信息和数据，以及经过统计分析得到的综合数据，可以通过各种形式表现出来。主要的表现形式有三种：查询结果、报告和图表。以缺陷的数据为例，可以查询特定人员提出的缺陷、特定人员解决的缺陷、所有当前提出的缺陷、所有当前处理中的缺陷，等等。可以生成本月缺陷情况报告，包括本月提出了多少缺陷、本月解决了多少缺陷，等等。可以用分布图、趋势图、时长图来直观地表现缺陷的总体情况。详细内容见 9.5 节。

11.4　配置审计

配置审计（Configuration Audit）的定义如下：

“An audit conducted to verify that a configuration item conforms to a specified standard or requirement.”

“执行审计以验证配置项符合特定的标准或需求。”

审计（Audit）这个词，被广泛地使用着。在软件开发领域，它的一个常见含义是，工作是否符合制定的流程、规程和规范。比如，发现缺陷的时候，是否按流程在缺陷跟踪工具里填了单子。再如，开发人员的修改在提交时，是否按要求写了注释。这个含义的审计，不是配置审计定义中所说的审计。尽管软件配置管理在这个意义上的审计也是不可缺少的。

配置审计定义中所说的审计，是指对象本身是否符合需求或相关的标准，而不是制作对象的过程。比如，设计一个扳手，S4 内六角扳手，那么，它一定要满足一个标准：其六角形对边尺寸为 4mm[3]。而如果客户订制一批扳手时，提出手柄要用红色的橡胶，那么将来给客户之前还要查看一下，手柄是不是红色橡胶的。

而软件配置项的审计，具体是指什么呢？

首先，是要确保需求文档反映了客户的需求，设计文档反映了需求文档的要求，源代码实现了设计文档的目标。这样一层层递进，以确保跑起来的程序，就是客户想要的。这些主要是通过评审来实现的，特别是在瀑布模型中。在确定需求的时候，要与客户一起讨论、评审，直到需求文档被一致认可。在确定设计的时候，要与相关人员一起讨论、评审，直到确定设计文档的内容能够实现需求文档的要求，并且能够编程序实现它。在源代码编写完毕，要做同行评审，看是否符合设计，是否实现了需求，是否正确无误。一般来说，只有通过了评审的软件资产，才能够供其他人使用。

其次，在源代码写完之后，要进行单元测试；集成时，要做粗略测试；进而由系统测试团队进行系统测试；最后，由用户或用户代表进行接收测试。这些，都是为了达到这样一个目标：软件配置项符合特定的标准或需求。同样，

[3] 不只这一个要求，详见国家标准 GB/T 4393。

只有达到了一定质量标准的软件资产，才能够作为开发人员继续工作的基础；只有达到了一定质量标准的软件产品，才可以发布给客户使用。

而从变更管理的角度，同样需要审计。每一个变更请求，在对资产完成相应的修改之后，通过系统测试来进行验证，证实请求的内容确实已被实现。在送交系统测试的每个程序版本中，通过系统测试来进行验证，证实所有本次版本应该实现的变更请求，其内容都确实已被实现。而在送交系统测试之前，通过查看变更集的描述等评审方法，确定所有本次版本应该实现的变更请求，都有对源代码的实际修改相对应，这也是在做审计。

上述审计，大都需要审计者具有软件所在具体领域的知识，对软件的具体功能进行检查。这又被称为**功能审计**（Functional Configuration Audit，FCA）。与之并列的，是**物理审计**（Physical Configuration Audit，PCA）。软件配置管理中的物理审计，是指对配置项的“外观”、位置等方面的审计。这些物理审计，就是软件配置管理工程师或者质量保证工程师要做的了。

配置项应该以合适的“外观”，出现在合适的位置。“外观”指的是配置项的名称、配置项的版本命名等，这些要符合规定，符合当初的计划；而位置指的是配置项的存储位置，这也要符合规定，符合当初的计划，它应该在正确的位置出现。这样的要求，是为了有一个好的软件开发环境，让大家能够方便地取到自己想要的东西。同时，也是为了保证，将来发布给客户的内容不丢东西。比如，如果发布给客户的内容是一个安装程序、一个发布说明文件和一个使用说明文档，那么在打包发布前进行检查，看这三项内容是不是都有，就属于物理审计。

11.5　一些软件研发模型中的配置管理

在**能力成熟度集成模型**（Capability Maturity Model Integration，CMMI）中，对包括软件配置管理在内的配置管理工作，从如下角度进行了划分。

- **建立基线**

首先，识别配置项；接着，建立配置管理系统，用来存放配置项；最后通过评审或测试后，由配置项组成基线，作为未来开发的基础。

- **建立并控制变更**

要追踪变更请求，这包括新功能、功能增强，也包括缺陷。要评估它们，分配给合适的人去处理它们，还要检查以确保它们确实被处理了。要控制对配置项的变更，如果要改它，需要合适的人同意。改好后，要适当检查，才能入库。

- **建立完整性**

首先，要对配置管理活动做足够的记录。比如，对配置项的修订历史，对变更请求的状态转换过程，对不同基线的差异，等等。其次，要进行配置审计。确定配置项的内容是否合适，并出现在合适的地方，确定基线的内容正确。

这三方面，是针对配置管理的要求。还有两方面，适用于各方面的管理，配置管理也需要。它们是：

- **制度化已管理过程**。

在一个软件研发项目中做配置管理，首先要建立配置管理计划，然后确保有足够的资源，包括工具、环境，也包括人员。在配置管理系统运转过程中要适当监控。

- **制度化已定义过程**。

要形成可以指导现在和未来多个软件研发项目的配置管理过程规范。这样的规范不是一成不变的。要收集相关的信息、数据和反馈，并基于此进行软件配置管理的持续改进。

在**软件能力成熟度模型**（Capability Maturity Model for Software，SW-CMM）

中，对软件配置管理从五个方面进行考查。它们分别是[4]：

- **执行约定**

在软件研发中，应该遵循一份描述软件配置管理大政方针（Policy）的文档。

- **执行能力**

要有相应的人，使用相应的工具和环境，来做相应的事，这些都要准备好。

- **执行的活动**

在进行了软件配置管理的计划之后，完成配置识别、配置控制和配置审计等各方面工作。

- **度量和分析**

进行软件配置管理方面的度量，并将度量结果用于确定软件配置管理活动的状态。

- **验证实施**

这里讲的是，软件配置管理工作，需要有某种形式的领导和监督。

对于 CMM、CMMI，在业界存在着激烈的争论。有鼓掌欢呼，热烈拥抱的，认为它能够“促进软件开发过程的改进，从而能够按时地、不超预算地开发出高质量的软件”；有抱怨连连[5]或尖锐讽刺的，认为 “CMM 是软件企业的耻辱符”[6]。

在 CMM、CMMI 之外，在 Rational 统一软件开发过程（Rational Unified Process，RUP）中，也涉及软件配置管理。这被分为计划项目配置与变更管理、

[4] 自 2005 年底起，SEI 已不对 SW-CMM 进行支持，SW-CMM 被 CMMI 取而代之。本文只对 SW-CMM 中软件配置管理部分做简略介绍。

[5] 一个典型的真实的抱怨：“我们公司……本身规模比较小，开发部就那么 10 来个人，结果为了过 CMMI3 搞得鸡飞狗跳，拼命地补文档……实际上反而降低了工作效率、延长了工期、提高了开发成本”。

[6] Tom DeMarco，《人件》和《最后期限》的作者。这是他在接受《程序员》杂志的采访时说的。

创建项目配置管理环境、管理变更请求、监控和报告配置状态、变更和提交配置项，以及管理基线和发布六部分。

这些软件研发模型和标准都有光辉的思想在里面，都是有价值的东西。然而，是充分发挥出价值，还是没有产生价值，甚至产生负面影响，要看怎么去用它们。基本的原则是，深刻领会标准的内在思想，结合企业的实际情况应用这些思想。而应用标准需要有所选择和裁剪，对于特定研发背景，要弱化甚至舍弃不适用的部分，要以合适的方式具体化和实现需要的部分。应用标准来不得教条，不要为标准而标准，更来不得表面文章。

这有点像音箱，挑音箱的时候，要掂一掂份量，份量沉的，比较让人放心。因为好音箱用料实在，好材料也大都比较沉。但这并不意味着沉的音箱一定质量好，因为可能是音箱里放了半块砖头。放半块砖头，那叫蒙事儿。

下篇　复杂场景中的配置管理

12

第 12 章　用分支实现交迭

是不是发布补丁版本这个词听起来比交迭这个词更熟悉一些？发布补丁版本是交迭的一种。让我们从这里开始谈起。

12.1　补丁版本

绘图产品 SuperPen 1.0 版，经过六个月的开发，终于上市！在大笔捞钱的同时，公司正在组织开发 2.0 版，引入更炫的功能，卖出更好的价钱！研发团队全体，继续日夜奋战。在一派大好形势下也有些不和谐的声音，一些用户抱怨，1.0 版里有这个 Bug、那个 Bug……虽然公司解释，这些都会在 2.0 版里修复，2.0 版会以优惠的价格卖给老用户。但是，有的用户坚持认为，应该免费提供对 1.0 版中的 Bug 的修复，并且现在就要提供，等不到三个月后 2.0 版啦。“好吧，好吧，”公司管理层决定，“我们出个 1.1 版，只包括对 1.0 版中的 Bug 的修复，不包括任何新功能。把 1.1 版免费发放给购买了 1.0 版的老用户。他们可别想趁机占便宜，得到 2.0 版里的新功能。”

开会回来，研发经理犯了难。“2.0 版的研发工作都开始了一个多月了。已经为实现新功能做了很多新的修改。难道这些都作废，退到 1.0 版从头来？”配置管理工程师却胸有成竹，“头儿，我们用那种叫分支的方法，就可以实现 1.1 版和 2.0 版同时开发，哪个也不耽误。”

没错，现在是分支大显身手的时候了。先考查一下现在的情形：主干上比较古老的部分，是目标为 1.0 版的研发；主干上某一点，是 1.0 版本发布了。随后，主干上开始了目标为 2.0 版的研发，现在正在向前演进。

为支持 1.1 版，我们从主干上 1.0 版本发布对应的那一点起，做一个分支，岔出来。这条分支的目标是 1.1 版，也可能将来 1.2 版也在这上面。我们把这条分支命名为 1.x。在这条分支上，我们修复 1.0 版用户发现的各个 Bug。一个又一个关于 Bug 修复的变更集被提交到这条分支上。分支在向前演进。在经历了若干次集成、测试之后，1.1 版在这条分支上正式发布！

把 1.1 版称做**补丁版本**（Patch Release），是因为 1.1 版跟 1.0 版相比，没有

什么大的变化，主要是一些修修补补。注意补丁还有另外一个含义，不完整的发布。比如说，安装某软件，在用 100MB 的安装包安装后，再安装一个 2MB 的汉化补丁，把该软件的菜单都改成方块儿字[1]。这个意义上的补丁，不是这里讨论的主题。

12.2 复用另一条分支上的改动

为 1.1 版所做的工作和为 2.0 版所做的工作，其工作目的不同，其代码修改不同，因此用两条分支来体现。然而，为什么一定要用分支的方式来实现，而不是用完全独立的两个目录或两个版本库来实现呢？主要原因是，分支除了有隔离的功能外，还对共享有比较好的支持。使用分支，便于彼此同步和复用。

理想的情形是，在 1.x 分支上，从 1.0 版起到 1.1 版之间所有的代码改动，都应该拿到主干上，贡献给 2.0 版。对于这种情形，可以使用版本控制工具提供的简单命令，一股脑儿地把改动从 1.x 分支合并到主干上来。

以上说的是理想的情形，1.x 分支上的全部改动都应该拿到主干上。在实际项目中，并不是这么理想的情况。比如有个 Bug，在 1.1 版上的修复，只是临时性的修复，治标不治本，因为治本的代价太大，需要改动软件中的一些底层结构。而在 2.0 版中，软件的相关底层结构已经优化，因此，这个 Bug 在 2.0 版中已经不存在了。

对此，比较保险的解决办法是，对 1.x 分支上的每一个变更集，均由开发人员做出判断，是否适合合并到主干。挑挑拣拣，合适的拿，不合适的不拿（见图 12-1）。

[1] 还有一类补丁，软件开发商们深恶痛绝…… 这类补丁的具体实现方法，显然也不是本书要介绍的。

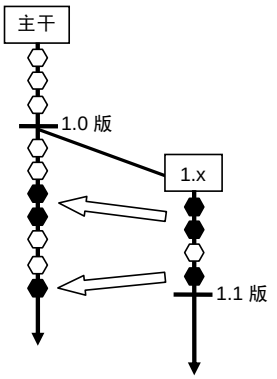


图 12-1　有选择地合并

这是以变更集为基础的挑选工作。而在有些团队里，使用的是以变更请求条目为基础的挑选工作。这里的变更请求主要是缺陷修复请求，可能还有少量增强请求。考查所有目标为 1.x 分支的变更请求，看看是否同样适用于主干。如果适合，就把相应的变更集从 1.x 分支复制到主干。

有选择地合并比较保险，但其管理成本也是显然的。在有些项目中，大家宁可容忍把没必要合并的改动合并过来，也不愿花费可观的精力去把需要合并的改动挑出来。于是就会使用把 1.x 分支上的所有改动都拿过来这样的合并方法，就好像工作在理想情况下。

上面讨论的都是从 1.x 分支到主干的合并。可能还存在相反方向的合并。比如，如果由于某种原因，一个缺陷先在主干上修复了。不久又决定要在 1.x 分支上使用同样的方式修复该缺陷。这时，就需要个案处理，把对应的变更集从主干复制到 1.x 分支上。注意这个方向上的合并不应该成为主流，两边都有的缺陷通常应该在 1.x 分支上先修复，因为 1.x 分支邻近发布，对缺陷修复的需求更紧急。

不论是以上哪种合并方法，执行合并操作时，一般来说，大部分改动可以自动合并到另一个分支上。当然，合并可能产生冲突，比如说某个文件在两个

分支上都改动了同一行，且改为不同的内容。这就需要人来解决，懂得源代码语义的人来解决。如果合并是由集成工程师或配置管理工程师主导的，那么出现合并冲突时，一定要找相关开发人员来解决，不要自己改代码。

除了代码合并冲突，还可能在编译时或运行时出现问题。这些问题的本质原因都是，在一条分支上的代码修改，是以当时这条分支上的源代码为基础的修改，它不一定适合放在另一条分支的末端的源代码上。如果这样的问题经常冒出来，严重干扰了目标分支上的代码质量，那么就要考虑先对合并过来的改动进行一定的质量保证工作，再最终提交到目标分支。而如果这类问题不多，那么依靠在目标分支上的（狭义）集成中或集成后的质量保证工作来发现和解决问题就可以了。

什么时候问题不多，什么时候问题多？一般来说，新的分支刚创建的时候问题不多，因为两个分支上的内容差不多。随着时间的流逝，两个分支上的内容相差越来越多，这时候把一个变更集从一个分支复制到另一个分支，就会越困难，出现各种问题的可能性就越大。可能需要因此而调整合并的策略，引入更多的检查。

关于不同分支间合并，还要注意合并的频率。如果合并工作一直拖着，过了很久之后积攒下大量的合并一起做，那么合并就会变成一件头疼的事。一团乱麻，出了问题不好解决。这也是因为大家已经对很久以前的改动不熟悉了，需要更多的时间回忆起来。可以考虑制定流程，每当提交一个变更集到 1.x 分支后，就要考查是否应该把它合并到主干。当然，定期地合并也不错，只要两次合并的间隔不要太久。

我们这里谈论的分支，可以是广义上的分支的不同具体形式：可以是狭义的分支，也可以是分布式版本控制工具中的不同的版本库，或者其他的方法。只要这些方法能够方便地做到不同（广义）分支之间的隔离和共享就可以。

12.3　甚至早在 1.0 版发布之前

1.1 版和 2.0 版两个版本可以并行地开发和发布。在前面两节中，这种同时开发多个版本的情况，发生在 1.0 版发布之后。那么，有没有可能在 1.0 版本之前呢？当然有可能。

假定此时，SuperPen 1.0 版本刚刚基本成型，也就是说，它的主要功能基本完成，大致能跑起来。可是，还有边边角角的少量功能没有完成，还有一些缺陷等待测试团队发现，等待开发团队改正。所以说，现在还不是对外正式发布的时候。胜利在望，公司老板决定到开发团队里去鼓鼓劲儿。可是到开发团队里巡视一番后，他发现，除了少部分开发人员还在开发新功能外，大部分开发人员都很悠闲。他们只是在测试团队报告发现缺陷的时候，三下五除二地改一改，而在其余的大部分的时间里，都是在上网、聊天，或者对着屏幕敲手指头发呆。

这样的情况当然不能允许，得给他们再找点活儿做——不等 1.0 版发布了，现在就启动 2.0 版的开发！

借助分支，可以做到这一点（见图 12-2）。

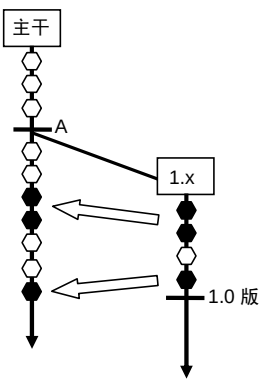


图 12-2　并行发布

假定当老板做出决定的时候，是在 A 这个位置，那么从 A 开始，创建一条新的分支，定名为 1.x。在这条分支上，开发少量新功能；在这条分支上，测试人员寻找并报告缺陷，开发人员修复缺陷。直到有一天，项目经理说，“嗯，已经足够好了，发布！”，于是 1.0 版在这个分支上发布。

而在主干上，从 A 那个位置开始就开始开发 2.0 版对应的新功能了。这跟分支上 1.0 版的开发互相隔离，不会彼此干扰。而另一方面，分支上的工作，包括少量新功能和大量的缺陷修复工作，对于 2.0 版的开发同样有用。所以，分支上提交的绝大部分的变更集，应该被复制到主干。这跟补丁版本的开发是差不多的道理。

那么，应该在什么时候创建 1.x 分支呢？当开发人员要提交第一个仅针对 2.0 版而不适用于 1.0 版的变更集的时候，这时主干的断面可以作为 1.x 分支的起始版本。此后，当开发人员要提交第一个对 1.0 版有用的变更集的时候，这时 1.x 分支应该已经存在，作为开发人员提交该变更集的目的地。如果上面的叙述理解起来比较绕的话，也可以大致描述为：当开发团队为 1.0 版和 2.0 版分头开发并准备提交的时候，需要准备好这两条分支。

真正的问题是，何时开发团队应该为 1.0 版和 2.0 版分头开发呢？晚一些开始，那么分支存在的时间短一些，同步和合并工作少一些。因此如果项目对人力资源、对成本比较敏感，那就晚一些开始。

而另一方面，如果早一些开始针对 2.0 版的开发，2.0 版就能够早一些发布。所以，要是项目对发布日期比较敏感，又舍得为此多配些人、多花点钱，那就早一些开始。

不论是早一些，还是晚一些，甚至是像 12.1 节里描述的那样，在产生了 1.0 版本之后再创建 1.x 分支，它们都被称做**并行发布**（Parallel Releases）。

12.4　甚至为了重要的内部版本

我们已经了解到，使用分支可以支持不同对外发布版本的开发。事实上，即便是内部版本而非发布给最终用户的版本，也可以考虑使用分支来支持。这和项目的开发模式关系很大。

假定 SuperPen 1.0 这个项目是这样的一个开发模式：在开发阶段有两个重要的里程碑，其中一个里程碑是内部版本 0.5 版，共有 50 个特性要实现；另一个里程碑是内部版本 0.8 版，新添 25 个特性，共 75 个特性。最后是对外发布版本 1.0 版，新添 15 个特性，共 90 个特性。除了要完成相应的特性列表外，0.5 版、0.8 版、1.0 版都要满足一定的质量要求。比如对于 0.5 版，要求无致命缺陷，严重缺陷小于 2 个，一般缺陷小于 20 个，轻微缺陷小于 200 个。

可以想见，在这样的一个开发模式下，两个重要的内部版本，0.5 版和 0.8 版，会得到相当大的关注。为了使其质量能够通过里程碑的要求，开发团队可能会做一些缺陷修复工作，以求按时通过里程碑，项目不至于被终止。与此同时，有些开发工作，尽管不属于此次内部版本的范畴，但也应该提早进行，以便尽早发布最终版本。这些开发工作，就与为通过某个里程碑而进行的缺陷修复工作并行了起来。如果这两类工作共用一个分支，那么新的特性所带来的缺陷，可能会使得内部版本达不到里程碑的质量要求。在这种情况下就值得使用两个分支（见图 12-3）。

在主干上，在 A 这个位置创建一个分支，命名为 0.5。在这个分支上，开始目标为 0.5 版的这轮测试，并不断提交缺陷修复的变更集，还不断形成新的送测版本，直到最后，产生 0.5 版。

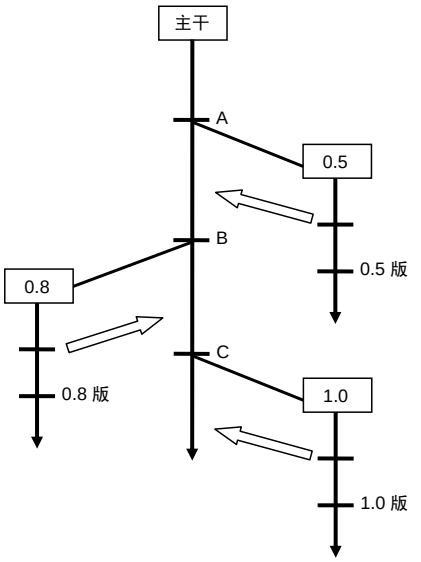


图 12-3　为了重要的内部版本

在为 0.5 版或 0.8 版本而创建的分支上，大概会发生多次系统测试。开发人员修复缺陷，然后集成工程师集成以生成基线。测试团队对其中的一些基线进行测试。一般来说，每次测试，测试团队会发现若干缺陷，反馈给开发人员，开发人员修改这些缺陷。集成工程师再一次集成，生成又一个测试版本，交给测试团队。于是，测试团队进行又一次测试。如此反复，直到能发现的缺陷基本都发现了，当下必须解决的缺陷基本都解决了为止。

那么，哪些缺陷是“当下必须解决”的呢？主要有两方面。一方面是，为了达到一定的质量标准，以通过项目里程碑，必须把重大的缺陷解决掉，同时，次要的缺陷的数量也不能太多。另一方面是，有些表层的缺陷，使得一些深层的测试无法进行，测试不到深层的缺陷。这样，就必须先解决表层的缺陷。比如，如果手机尚发不出短信，就没法测试接收方收到的表情符号是否显示正确。

在 0.5 和 0.8 这两个分支上的变更集都是缺陷修复，对程序改动不大。并且，这两个分支存在的时间都不长，和主干的差距都不大。因此，一般来说，提交

到它们上面的变更集都应该被复制到主干上，并且是相当自动化的合并，一般不用人工干预。通常，这样的分支合并工作由集成工程师统一完成就可以了。这可以是每天定时作一次，或者发生在每次送测版本形成的时候，统一把本次送测版本中新提交的变更集合并到主干上。

在上面的例子中，我们先描述了一个开发模式，在这个开发模式下，适合为内部版本开分支。还有些其他的开发模式也适合为内部版本开分支。比如，有些产品在发布给最终用户之前，必须通过某个权威机构的测试，而这个测试并非要求产品的全部功能，只是关注于这个权威机构相关的一些功能及其质量。在这种情况下，也值得考虑为此创建一个单独的分支，防止其他功能的开发对它所关注的质量的干扰，及早通过权威机构的测试，以便启动其他若干只有通过这个测试才能开始开展的工作和流程，为将来出货做准备。在这个例子中，由于使用了分支而赢得了宝贵时间[2]。

12.5　甚至为了每次集成

在第 4.3 节讲解（狭义）集成的时候，我们讲到，当集成过程中出现构建失败等问题的时候，需要修改源代码，需要新的变更集。而另一方面，又不希望与此无关的变更集溜进来，因为存在着带来新的问题的风险。

为此，你可能会考虑在集成开始的时候把主干冻结，不允许新的提交，除非是与修复集成问题相关。这样做倒是安全了，却牺牲了效率。假如集成工作不需要多少时间，那么开发人员也就忍了——至少，他们是可以在本地修改代码的，只是不让提交而已。但是，如果系统集成工作需要经过比较长的时间，抱怨就会很大。开发人员新修改的代码迟迟不能提交，就会影响他们互相之间取得彼此新的工作成果。这时，他们可能会暗度陈仓，把新修改的源代码文件或目录通过电子邮件等方式相互传递。这就退化到了没有版本控制工具时候的情

[2] 当然，最终发布版本，也必须是可以通过该测试才行。

形，这很危险。可是，开发人员会说，“没办法，我们等不起。”

当系统集成预计需要比较长的时间的时候，可以考虑让集成—问题解决工作与开发人员的继续开发和提交并行起来。这仍然是靠分支实现的（见图 12-4）。

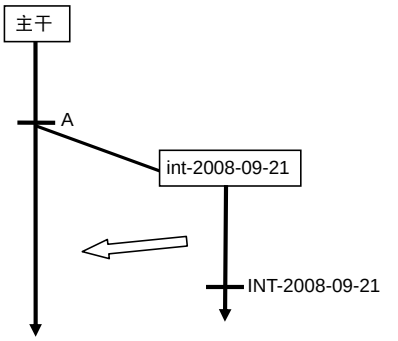


图 12-4　为解决集成问题开分支

在 A 点，集成工程师开始做集成工作。他首先在 A 点打一个标签，接着，取 A 点的整体版本到本地的工作区。然后，他尝试着编译、链接、打安装包和粗略测试。如果顺利，那自然好，直接把安装包保存好，告诉测试团队就行了。而如果不顺利，那就为解决本次集成的问题而创建一个分支。在图 12-4 中，分支的名字是 int-2008-09-21。

在该分支上仅仅为解决集成上的问题而进行改动。这可能是集成工程师完成的，也可能是开发人员完成的。不论由谁完成，都是首先把工作区定位到集成分支上，然后修改相应源代码，测试确信没有问题后提交。在这之后，集成工程师重新做一遍集成操作：下载分支上的最新源代码，编译、链接、打安装包，然后粗略测试。如果都没有问题，完成集成，生成基线，并将安装包保存好，以供开发人员和测试人员使用。同时，集成工程师把集成过程中的修改合并到主干上（见图 12-4）。

这个方法比较适合于每次集成所需时间比较长，而主干末端的质量又经常

有问题的情形。这种情况下，如果每当集成出问题时，不使用分支，而是提交问题的修复到主干末端并进行新的一轮构建和粗略测试，则很有可能又因为新的问题而失败。这些新的问题是由上次失败的集成对应的版本之后提交到主干的变更集所带来的。

而如果每次集成所需时间比较短，主干末端也不经常出问题，则这个方法并没有太大必要。在这种情况下，每当集成出问题时，可直接提交问题的修复到主干末端并进行新的一轮构建和粗略测试，在大多数情况下，新的一次集成将成功。

考虑用这个方法的时候，还要注意所在开发团队使用的版本控制工具的支持情况。如果基线是在用于解决集成问题的分支上，而不是在主干上，那么开发人员从基线更新可能不太方便。在将来提交时，还要注意是提交到主干，而非基线所在的解决集成问题的分支。

12.6　双分支结构

在上一节，为了解决集成遇到的问题，同时不阻滞开发人员继续提交新的修改，我们开出分支。每当集成遇到问题的时候，我们就开出分支。这样分支会不会有点多？

有办法改善，可以用一条长期生长的分支而非很多个短分支来支持集成过程中的修改。并且，总是在该分支上发布新的基线。由于这个原因，该分支常被称做发布分支。

具体来说，开发人员提交代码改动到主干。每次集成时，集成工程师把主干上的最新内容合并或干脆复制到发布分支。在发布分支上，下载源代码、编译构建、粗略测试。如果没有问题，在发布分支上用打标签等方法制作正式的基线，供开发人员和测试人员在必要时使用，甚至发布给外部用户。而如果编

译构建或粗略测试时发现了问题，就在发布分支上修改代码解决这些问题，然后再生成基线。当集成完成后，要把发布分支上为生成本次基线所做的代码修改合并回主干。

可以看出，在集成的过程中，主干一直开放给开发人员，随时提交任何修改，与本次集成互不干扰。这和使用多条短分支来支持集成的效果是一样的，却能避免频繁开分支。

为什么这里可以用一个长分支代替多个短分支呢？因为总是在一次集成结束之后，才开始下一次集成，因此用同一个分支支持多次集成不会打架。

我们姑且把这种一个分支与主干（本质上也是分支）长期并行并且上面的内容彼此很接近的分支结构，称做双分支结构。这种结构也同样可以用于其他的场景。下面以补丁版本为例介绍。

A 公司是一个大型金融公司。企业内部有个软件研发团队，他们研发某款软件，部署到服务器上，专供 A 公司自己内部使用。在这种情景下，A 公司作为唯一的用户，在某一时间只会使用软件的一个发布版本，而非多个。并且，软件发布通常是循序渐进的，每次增加一点新功能。

在这种情况下，可以使用双分支结构，即主干加发布分支。在主干上，开发人员提交，集成工程师集成，产生基线。该基线被合并甚至是复制到发布分支上，向 A 公司内部发布，并部署到服务器。

当在服务器上运行的程序暴露出比较严重的缺陷需要紧急修复时，当因为公司业务政策的调整而需要紧急调整某个功能的算法时，需要紧急开发和发布一个补丁版本。这时，补丁版本的开发可以在发布分支上直接完成，并在发布分支上产生基线，向 A 公司内部发布，并部署到服务器。此后，把发布分支上为此次补丁版本所做的修改合并回主分支。

当发布分支上为补丁版本忙碌的时候，主干上正在开发和集成下一个常规版本。两者并行进行，互不干扰。这和每次补丁发布都创建一个新的短分支，效果是一样的，却能避免频繁开分支。

为什么这里也可以用一个长分支代替多个短分支呢？因为总是在一次常规或补丁版本发布后，才会有下一次常规或补丁版本的发布，并且这些发布总是不断向前演进的。因此用同一个发布分支支持多次发布不会打架。

12.7　都是交迭

交迭（Overlap），就像房顶上的瓦。瓦片交迭在一起，一块遮住另一块的一部分，这样从屋脊到屋檐。试着把自己想象成一只猫，从屋脊顺着瓦的方向往下走。你会看到，在一块瓦上还没有走到头的时候，新的一块瓦已经搭在正踩着的瓦下面了。

在 12.1 节里，正踩着的那片瓦是目标为 1.0 版、1.1 版、1.2 版等 1 系列版本的研发，而前面新的一片瓦是目标为 2.0 版等 2 系列版本的研发。正踩着的瓦，到了 1.0 版这个位置，还没有到终点的时候，新瓦，也就是目标为 2.0 版的研发，已经搭在下面了（重画于图 12-5）。

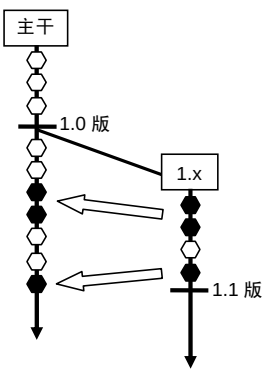


图 12-5　补丁版本

在 12.3 节里，正踩着的那片瓦，还是目标为 1.0 版、1.1 版、1.2 版等 1 系列版本的研发，而前面新的一片瓦，还是目标为 2.0 版等 2 系列版本的研发。不过，这回相互交迭的，要更多一些，正踩着的瓦，到了 A 那个位置，还没到 1.0 版那个位置，新瓦，也就是目标为 1.0 版的研发，就搭在下面了（重画于图 12-6）。

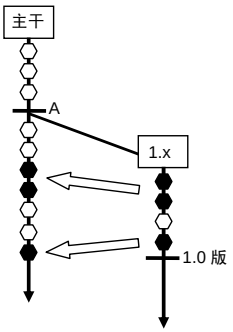


图 12-6　并行发布

在 12.4 节里，正踩着的那片瓦，是目标为 0.5 版的研发；而前面的新的一片瓦，是目标为 0.8 版的研发；正踩着的瓦，到了 A 那个位置，还没有到 0.5 版那个位置，新瓦也就是目标为 0.8 版的研发，就搭在下面了（重画于图 12-7）。

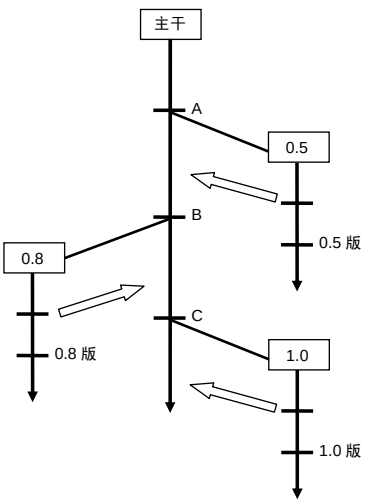


图 12-7　为了重要的内部版本

在 12.5 节里，正踩着的那片瓦是目标为 INT-2008-09-21 版的开发和集成，也许将来还有目标为后续版本的测试—缺陷修复。而前面的新的一片瓦也许是 INT-2008-09-28 版的开发和集成，如果是每周集成一次的话。正踩着的瓦到了 A 那个位置，还没有到 INT-2008-09-21 版的那个位置，新瓦，也就是 INT-2008-09-28 版的研发，就搭在下面了（重画于图 12-8）。

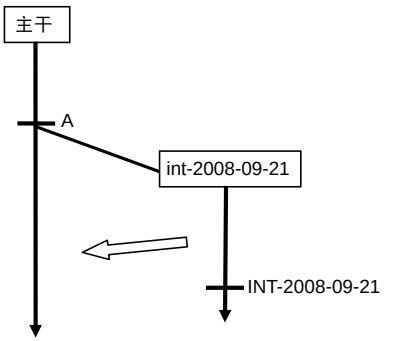


图 12-8　为解决集成问题开分支

所以说，在本质上它们都是交迭。用分支支持交迭，一条分支上，重点是质量，重点是稳定性。另一条分支上，是要增加新功能，是要全力发展。两条分支，既防止了后者干扰前者，也防止了前者阻滞后者。

如何处理好发展与稳定的关系？用分支啊。

13

第 13 章　用分支实现变体

兄弟姐妹，他们之间既有共同的基因，也有彼此的区别。在软件世界也一样，一些软件产品间，是类似于兄弟姐妹这样的关系。

13.1　什么是变体

变体（Variant），又译为变种[1]。它是指这样一些软件产品，它们彼此有一些相同之处，但又彼此有所区别，就好像一群兄弟姐妹。弟弟将来无论怎么长，也不会长成跟哥哥现在的样子一模一样；姐姐像妹妹现在这么大的时候，跟妹妹现在的样子也不完全一样。所以说，他们之间的差异是与生俱来的、本质上的，并不完全是时间演进和个人成长的结果。变体也是一样。所以 SuperPen1.0 for Windows 版和 SuperPen1.0 for UNIX 版之间互为变体。而 SuperPen 1.0 版和 2.0 版之间并不是互为变体，尽管它们之间也相似。

那么，什么情况下会产生变体呢？

- **因支持不同操作系统而产生变体**

这是产生变体的第一种原因。打算在 Windows 操作系统上卖的产品，很有可能在 UNIX 操作系统上也有市场。但是，在 Windows 上运行的软件，常常是不能直接拿到 UNIX 上运行的[2]。这至少需要在 UNIX 上重新编译、链接、打包一遍。情况往往比这个还要复杂些，不仅要重新构建，连源代码本身都要做一些修改。比如，在 C 语言里，为获取当前本地时间，在某个操作系统上，需要调用 localtime 函数，而在另一个操作系统上，需要调取 time 函数。那么，就需要根据所在的操作系统，调用不同的函数。只是调用不同的函数还好说，如果程序的一部分，当年是在 Windows 操作系统上，用 Visual Basic 写的，那恐怕在 UNIX 系统上，就需要把这部分程序换种编程语言重新写了。

- **因不同的功能集而产生变体**

以微软的操作系统 Vista 为例，它有两大类共七个版本，即变体。分别是，家庭类的初学者版、家庭基础版、家庭标准版（Premium）和家庭终极版

[1] 考虑到这样的翻译可能会让人产生丰富联想，我们还是管 Variant 叫变体吧。
[2] 用 Java 语言编写的程序，一般是可以的。用 Perl、PHP 等语言编写的脚本，也常常是可以的。

（Ultimate）；商务类的 Vista 小企业版、Vista 专业版和 Vista 企业版。不同的变体，其功能有所不同。有的功能多些，有的功能少些。不同的变体，是针对不同的消费群体的不同的需要而设计的。

- **因客户定制而产生变体**

我们通常看到的，在市面上卖的，在柜台里摆着的软件产品，都是“标准”的。一个产品，会卖给成千上万个客户，每个客户得到的都一样。但是，还有不少软件产品，每个客户得到的是不同的，因为他们想要的东西是有区别的，比如，为电力系统监控开发的软件。由于不同地区的电力系统，其用于监控的硬件设备的型号是不同的，因而软件要为此进行特殊定制。不同地区的电力局，其内部的管理制度、管理流程是有区别的，因而软件也要为此进行特殊定制。通常，这样的一个面向特定用户的定制软件，会被单独立项。

以上三种产生变体的原因，还仅仅是举例。变体产生的原因有很多，但它们都带来一个相同的挑战，那就是，需要支持多个变体的开发和发布。

13.2 用分支支持变体

假定，基于 SuperPen 的标准版 1.0 版，开发 SuperPen 1.0-A 版。这是为客户 A 专门定制的一个版本，里面增加了一个只有客户 A 才需要的功能：点石成金。那么，为 1.0-A 版，建立一个分支，命名为 1.0-A。这个分支的起点，就是主干上的 1.0 版。建立分支后，在该分支上，完成客户 A 需要的特殊功能，即点石成金的开发。开发完毕，测试通过，就在该分支上形成了 1.0-A 版（见图 13-1）。而在主干上，继续开发标准版，2.0 版。不太难，是不是？

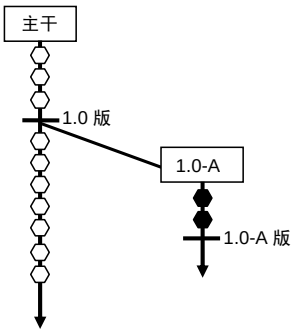


图 13-1　变体（简单情况）

依此方法开发，1.0-A 版必然在 1.0 版之后发布。如果点石成金这个功能比较复杂，那么 1.0-A 版的发布时间就会拖后很多。有没有可能让 1.0-A 版在 1.0 版发布之时或稍后一点就发布？可以。

在 1.0 版尚未完全成熟时，就可以创建 1.0-A 分支。此后，在主干上，软件功能日趋成熟，缺陷越来越少。而在 1.0-A 分支上，点石成金这个功能被逐步开发出来。

1.0-A 这个版本，不仅需要点石成金这个功能，而且需要 1.0 版的完整功能及其质量。因此，在主干向前演进的同时，主干上的改动要不断地同步到 1.0-A 分支（见图 13-2）。

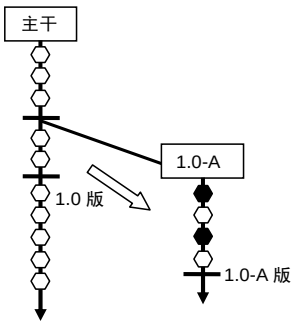


图 13-2　在发布前开出变体分支（简单情况）

可以看到，两条分支，很好地隔离了两种不同目的的工作。同时，由于使用分支，工作成果共享也方便，这体现为从主干到变体分支的合并。

13.3 随主干继续演进

假定客户请求[3]将定制的版本随着标准版本的升级而升级。即，当推出标准版 2.0 版时，客户请求提供 2.0-A 版，在保留点石成金功能的同时，去掉 1.0 版里发现的缺陷，添上 2.0 标准版里新增加的功能。这时候怎么处理？

方法之一是，继续用上一节所讲的思路，把主干上 1.0 版与 2.0 版之间所有的修改，都复制到分支上。如果是这样，那么分支的名字应该叫 A，而不是 1.0-A。因为，分支的目的，不仅仅是产生 1.0-A，还会产生 2.0-A，或许还有 3.0-A，等等。将主干上 1.0 版与 2.0 版之间的所有的修改都复制到该分支上后，集成、测试通过，产生了 2.0-A 版（见图 13-3）。

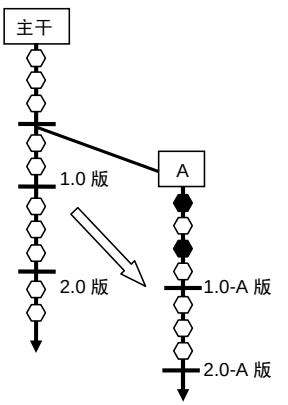


图 13-3 提升变体版本（方法一）

请注意，从标准版 1.0 到标准版 2.0，这之间可能有了相当大的变化。这些变化，很可能会影响到 1.0-A 中的那个点石成金的功能。所以，安全起见，一般应该对 A 分支上因为分支合并而引入的代码修改，进行某种浏览和审查，如有

[3] 说是请求，其实就是要求。真拿这些客户没办法……

必要，做适当调整，确保它们不会影响点石成金的功能。这可能工作量不小啊。有没有办法解决？

有办法解决，那就是用方法二（见图 13-4）。

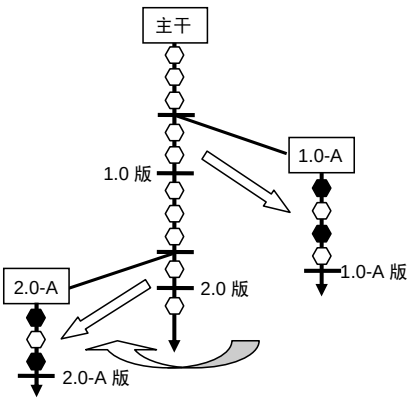


图 13-4　提升变体版本（方法二）

为产生 2.0-A 版，不是把主干上的内容合并到变体分支上，而是创建一个新的变体分支，把原变体分支上的为变体特殊需求所作的修改复制过来。具体地说，是在主干上 2.0 版或者 2.0 版之前的某个位置，创建一个新的分支，命名为 2.0-A，为产生 2.0-A 版服务。然后，把分支 1.0-A 上的所有的为变体特殊需求所作的修改所对应的变更集（图中黑点），都复制到分支 2.0-A 上。此外，如果是从主干上 2.0 版之前的位置开出的分支，那要把主干上的修改同步到这条变体分支上。这样，产生了 2.0-A 版。

用方法二，对分支合并引入的代码修改进行浏览和审查，就变得容易多了。因为，这次分支合并的内容，主要是点石成金这个功能对应的代码改动。源代码的改变量要比从 1.0 版到 2.0 版的源代码的变化少得多，浏览、修改起来方便。当然，这样做的前提是，对 2.0 标准版的架构和代码实现比较熟悉，知道合并引入的代码修改是否合适。

除了优点外，方法二也有弱点。如果需要频繁地根据标准版的最新发布版

本创建变体的发布版本，那就需要建很多分支，为变体的每一个发布版本服务。每一次，都需要浏览变体特定的所有修改。这代价也不小。当需要比较频繁地发布变体版本的时候，可考虑方法一和方法二结合使用。比如，为 1.1-A 版工作时，使用方法一，延用 1.0-A 版所在的变体分支。而为 2.0-A 版工作时，使用方法二创建新的变体分支。

上面讨论的都是理想情况。在理想情况下，公共的修改，包括公共的缺陷修复、公共的功能增强、公共的新功能等，都首先发生在主干上。而分支上的，都纯粹是针对客户特殊需要而进行的改动。这只是理想情况，而实际情况要复杂一些。在分支上的修改，可能不仅仅是关于点石成金的；可能有一些针对标准 1.0 版的缺陷修复，因为客户急着要的也在分支上完成了；也可能在分支上，应客户请求，追加了一个小的功能增强，而这个功能增强，稍后的标准版 2.0 版里也想实现它。

如果在实际的情况下，在变体分支上，不得不发生了这样的具有公共价值的改动，那么这样的改动应该被复制到主干。这体现为，把对应的变更集复制到主干。于是，这些改动就真正成为公共的改动了，真正发挥了公共价值。

还记得我们是怎么使用分支来支持交迭的吗？在那里，代码改动在分支间的主要流动方向是，从两边向中间流动，从分支流向主干。而当用分支支持变体时，代码改动在分支间的主要流动方向是，从中间向两边流动，从主干流向分支。这是分支支持交迭和变体时的重要区别。

一个产品的多个历史版本，体现的是软件的演进。为此开分支，是为了质量和稳定性，所以要合并回来。一个产品的多个变体版本，体现的是彼此并列的差异，是“永恒”的差异，不能合并到一起。反之，是公共的改动要传导到变体中去，让变体在必要时随着标准版本的演进而演进。

13.4　把公共改动放到主干上

使用分支来支持不同目标的工作之间的交流和共享，这是个好事儿。然而，仅仅引入了分支还不够，还要用正确的策略使用它。

某公司开发的软件，是行业应用软件，不是卖给普通老百姓，而是卖给某个行业里面的众多企业的。这些企业在购买的时候，常常提出若干功能需求，这些功能需求，还没有在软件已有版本里实现。所以，需要为特定客户进行单独立项，以进行特定开发，这样才能把产品卖出去。每一个这样的项目，用一种鱼的名字来表达。每一个项目，对应于一个分支，最终产生一个产品（见图 13-5）。

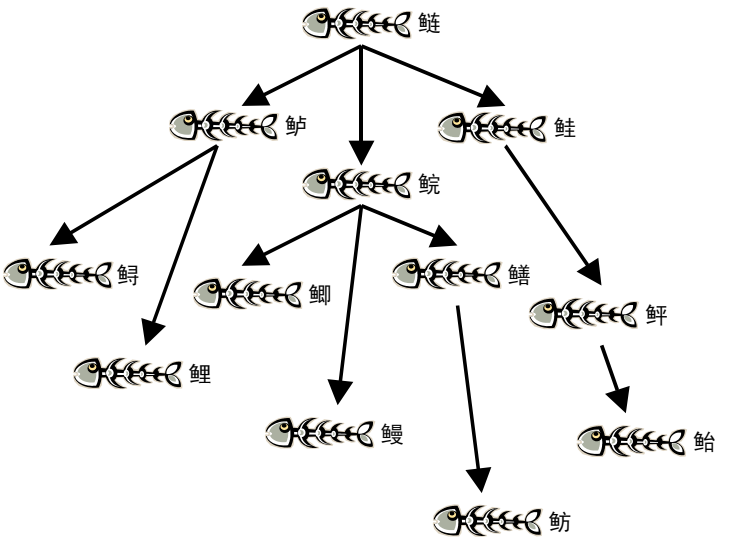


图 13-5　鱼的裂变

这是典型的自然形成的分支图。每个新的项目，都是从某个已有产品开始，开出一条新的分支，开发这个产品所需要的但还不具备的新的功能，直到发布。这种方式，开始的时候工作得还好，可是随着时间的推移，就越来越不对劲了。

以鲂为例，当计划开发鲂的时候，调查显示，它与鲤相比，需要再开发 50 个特性。与鳗相比，30 个。与鳝相比，20 个。与鲑相比，70 个。好吧，看来应该基于鳝继续开发。然而，这相差的 20 个特性，有 18 个已经在鲤或者鳗或者鲑上实现了。现在因为是基于鳝，于是这 18 个特性要再开发一遍。不甘心啊！不甘心……

是否可以通过变更集在不同分支上的复制来实现减少重复工作？会好一些，但仍然很麻烦。鲤、鳗、鳝、鲑四个项目之间，系统结构可能已经差别很大了。相同功能，其对应的源代码实现，不得不有很大的差异。另一方面，要操作的分支很多。这回是从鲤、鳗、鲑上扒出来有用的东西，贴到鳝上，下一个项目又要从哪些地方复制到哪儿呢……

问题出在哪儿？在前面这个例子里，每个新的项目，都是从某个已有产品开始，开出一条新的分支，开发这个产品所需要的但还不具备的新的功能，直到发布。注意这里的用词，“这个产品所需要的但还不具备的新的功能”，这并不意味着，这些都是这个产品所特有的功能。很可能不少功能或早或晚会被其他产品用到。这样的功能，本质上是公共功能，而非变体特殊的功能。

既然是公共功能，就应该尽量在主干上开发。这样，变体分支在创建时就已经获得了这样的公共功能。在变体分支创建后，也可以通过合并，从主干不断获得公共功能。

而如果因为某些原因，某个公共功能首先在变体分支上开发出来的时候，应该尽早把该功能对应的代码改动合并回主干。这样，其他变体就可以方便地使用该功能。

分支图应该规划为类似图 13-6 的样子。

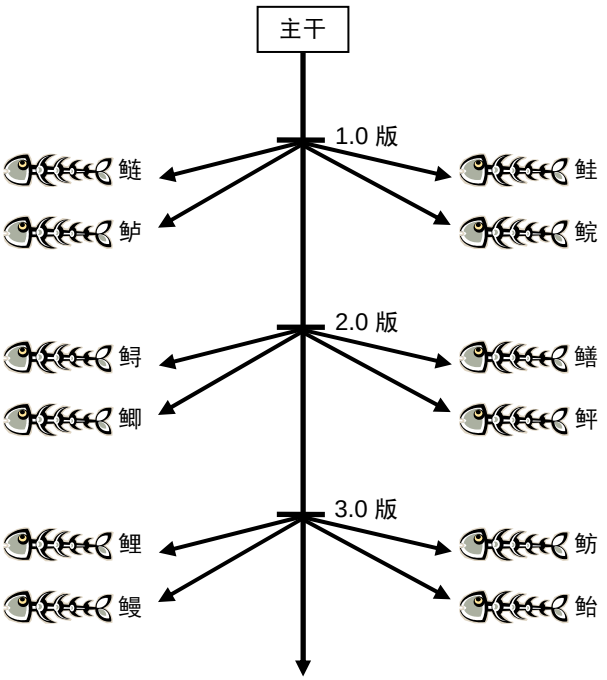


图 13-6　改进的分支图

13.5　变更管理对交迭和变体的支持

从某种意义上说，交迭和变体把一个产品变成了相互关联的多个产品。那么相应的，一个变更也就在某种意义上变成了相互关联的多个变更。假定，SuperPen 这个产品有四个主要发布版本，从 1.x 到 4.x，都在维护。再假定，SuperPen 这个产品有 5 个变体，从 a 到 e，也都在维护。那么，某个缺陷，可能会在 20 个版本里同时具有，尽管刚开始的时候，只是某个变体的某个发布版本上发现有这个问题。

可以使用一个主变更条目，及其相关联的 20 个子变更条目。主变更条目描述这个缺陷的现象以及严重程度等选项，因为这些信息在 20 个版本里（几乎）都相同。而每个子变更条目对应于某条分支。同样的缺陷，在某条分支上可能

已经修复了，但还没有同步到另一条分支上，而第三条分支上根本就不打算修复它了。因此，不同分支上，也就是不同的子变更条目，其状态可能不同。此外，各分支上，发现该缺陷的测试版本不同，缺陷修复所对应的变更集编号也可能不同，等等。这些不同分支上可能不同的信息应该放在子变更条目里。

这是一个复杂的解决方案。在实践中经常简化。极端的情况是，只用一个变更条目，在该条目中只记录一条分支上的情况。当该缺陷在该分支上修复后，该缺陷修复对应的变更集，随后会被复制到所有的分支上。于是，在那些分支上，该缺陷也就被修复了。

在很复杂和很简单两个极端之间，还有各种可能。比如，只用一个变更条目，但在该条目中，也适当记录多条分支上的情况，比如，在哪些分支上有这个问题，和/或应该在哪些分支上解决，和/或已经在哪些分支上解决，等等。

不论采用哪种方案，都是要达到或基本达到相同的目的，那就是，保证该缺陷在所有应该修复该缺陷的分支上，都被及时修复了。这可以通过在缺陷跟踪工具中的复杂的解决方案实现，也可以是通过版本控制工具中，分支间合并的管理实现。

此外，应该能够方便地查看到，某个缺陷是否已经在某条分支上修复了。同样的，这可以是通过在缺陷跟踪工具中的复杂的解决方案实现，也可以是其他方法，比如通过版本控制工具与缺陷跟踪工具的集成，根据某条分支上的变更集来推断出该分支上已修复了哪些缺陷。

13.6 历史版本与变体版本的命名

发布给客户的版本具有客户能看到的版本号。这个版本号应该是对客户有价值的信息。典型的是，三段式版本号，例如 2.1.3。其中第一段是主版本号，其升级标志着重大功能的变化。如果这个软件与其他软件配合工作，则可能由

于该软件提供的接口的变化而产生问题。主版本号从 1 开始。第二段是次版本号，其升级意味着新增或增强了少量功能。通常，即便是该软件提供的接口发生了变化，它也能兼容以前的接口定义，不会影响到它与其他软件配合工作。次版本号从 0 开始，并在主版本号升级后清零。第三段是修订版本号，其升级意味着又有一些缺陷被修复了。当然，前两段的升级也包含缺陷修复。修订版本号也是从 0 开始，并在次版本号升级后清零。

通常，每当主版本号升级，都意味着要出现新的分支。该分支与主干上的大规模开发相并行，致力于该主版本号的一系列版本的质量提高，直至发布。有时，也为次版本号从主版本号所在分支中再次开出分支。因为在某个次版本号临近发布、希求稳定的时候，可能有一些为下一个次版本号所作的少量开发工作正在进行中。绝少为修订版本号而再度开出分支。

以上讲的是 1.0.0 及其以上版本的常用命名规则，及其对应的分支策略。有意思的是，在 1.0.0 版本之前，一般采用另外一套版本命名规则。1.0.0 是第一个正式对外发布的版本。在此之前的非正式版本，第一段是 0。一般来说，第二段是从 1 到 9 之间的不断增加的数字。第三段是从 0 开始不断增加的数字，当第二段变化时，第三段清零。在 1.0.0 之前的版本号，其值越接近 1.0.0，代表着越临近正式发布。1.0.0 之前的分支策略，与开发模式有很大关系，详见第 12.4 节的论述。

以上是不考虑变体版本、只考虑历史版本的情况。当考虑变体时，可以肯定的是，变体版本要体现出，这个版本是为哪个变体服务的。而需要确定的是，应该给出对应的标准版本的信息，还是该变体本身的版本信息。前者很常见，比如 SuperPen1.0 for Windows 版和 SuperPen1.0 for Unix 版，它们的功能应该是几乎一样的，只是因为不同操作系统而产生的变体。后者同样常见，比如手机用户通常是关心他所用的这款手机，有没有新的版本可以用来更新自己的手机。他关心的是这款手机软件本身的版本，而不关心若干款手机的软件是否相互是

变体的关系，它们的开发是如何组织协调的，它们有没有标准版本，等等。[4]

一般来说，当变体关系对用户来说非常明显时，取标准版本号。当变体关系对用户来说不那么明显，用户不关心背后是否有什么标准版本或平台时，取变体本身的版本号，作为用户看到的发布版本号。当然，如果用户对两方面都关注，那就在发布版本号或发布说明中，把两方面都体现出来。

以上讨论的是对外发布版本的版本号。在开发过程中，作为每次（狭义）集成的成果，基线本身也需要版本号。有两个简单的方案。其一是，确定其目标发布版本，然后在目标发布版本的三段式版本号之后，再加上一段，作为序列号。该序列号从 0 或 1 开始。每次集成，序列号就加 1。当目标发布版本变化时，序列号清零。

另一个方案是，内部版本号包括两段，第一段是所在分支的名称，第二段是序列号，该序列号从 0 或 1 开始，每次集成，序列号就加 1。当分支变化时，序列号清零。

[4] Android 改变了这一规则。使用 Android 手机的用户，通常也会关心他所用的手机软件的操作系统的版本，比如 Android 2.3。

14

第 14 章　用设置实现变体

如何支持变体的开发？分支只是方法之一。另一类方法是通过不同的设置、通过不同的资源、通过不同的数据来产生变体。

14.1 在构建时读取设置信息

在版本控制工具看起来是相同的源代码，构建工具却可以选取其中不同的部分来构建，于是就会产生不同的程序，互为变体。

以 C 语言为例，对 C 语言的编译链接通常是通过 Make 工具完成的。Make 工具在执行时，会去查看一个叫 makefile 的配置文件，以决定如何编译，其中包括需要编译哪些源文件，等等。因此，尽管是从版本控制工具中取到相同内容，由于使用不同的 makefile 指导编译工作，可以编译不同的源代码文件集，获得不同的构建结果。

在编译链接时，甚至可以使用相同的 makefile。在 Make 工具使用 makefile 时，同时传递给 Make 工具不同的参数，也可以令 Make 工具选取不同的源代码文件集进行编译，获得不同的构建结果。使用这个方法，由于 makefile 相同，就可以把 makefile 和源代码文件一同放到版本控制工具中管理，很方便。

不仅可以选取不同的源代码文件，甚至可以选取一个源代码文件的不同部分。这就需要把宏定义写到源代码文件中。编译该文件时，当编译参数不同时，宏决定了编译该文件的不同部分。如果只在 makefile 上做文章，为不同变体选择不同的源文件，有些时候，尽管是公共的内容，但却需要重复写在不同的变体对应的不同的源文件中。而如果同时采用源文件内添加宏定义的方法，就可以在更细的粒度上分辨和控制哪些内容是变体特殊的，哪些内容是公共的。因此，上面所说的“重复”的问题，会大大减少。

有些时候我们会听到编译开关、编译选项这些词汇，说的就是通过对编译过程的设置而得到不同的可运行程序，也就是本节介绍的方法。

14.2　安装和运行时读取设置信息

有些应用程序，不论是运行时显示哪种语言，从网上下载到的，都是相同的安装包。而在安装的过程中，安装程序会询问用户，将来运行时显示哪种语言。如果你选择了中文，那么将来程序运行时，其菜单、提示信息等都是中文的。如果选择了英文，那就都是英文的。这是典型的通过安装时的设置，产生了具有不同的外在行为的程序，其效果就像是由不同的源代码构建而得的。

很多应用程序在运行时从配置文件中读取设置信息。配置文件里说启用某个功能，程序就具有这个功能。配置文件里说不启用，那就好像当初开发这个程序时，从没开发过这个功能。这是典型的通过运行时的设置，而产生了具有不同外在行为的程序。

在运行时读取设置信息，还有其他形式，比如在 Windows 操作系统中，可以把设置信息放到注册表里。还有可能，这些设置信息是加密的，用户不能修改，只能由软件的提供方提供或修改。这常发生在用户只购买了或只应免费享有某个软件的部分功能的时候。如果在这种情况下，用户可以轻易地通过修改设置信息而获得更多的功能，那软件公司不就亏啦。

运行时的程序还可能通过探测运行环境，而改变自己的外在行为。比如一款手机，当它运行在某个运营商的网络中的时候，它显示出该运营商所要求的一些功能。而换一个运营商，就连相关的菜单项都找不到。

在安装和运行时读取设置信息，与在构建时读取设置信息相比，最大的好处是减少了构建的数量。构建一次，即可产生多个变体。这不仅减轻了集成工程师的负担，也节省了开发人员的时间。

在运行时读取设置信息，可能会影响到程序运行的性能。与其他方法相比，程序可能需要更多的内存，程序可能运转得有点慢。这方面的影响通常很小。但在一些特定情况下，需要具体分析。

构建时、安装时或运行时读取设置信息，实现变体。这一类方法与用分支的方法相比，其好处主要是，不需要做代码合并工作。对于公共代码，在工作区中一次性地完成工作，即对所有变体都产生了效果。而如果某个公共功能的改动涉及对于变体特殊的源代码，由于在工作区中可以同时看到不同变体的特殊代码，因此可以方便地对特殊代码进行修改，把所有变体的特殊代码一次性地都改好。因此，对于一项开发任务，从整体上看，需要消耗的人日减少了。

而从另一方面讲，这也正是它的弱点。每次修改代码，都要考虑在所有相关变体上正确实现。这与仅考虑在标准版本上实现相比，需要更多一些的时间。包袱变重了嘛。因此，在人力资源充沛的情况下，对于一项开发任务，在主干上完成它所需要的最短时间变长了一些。

14.3 资源与数据

前两节我们谈论的设置，主要是谈论一些关键的参数，这些参数数量不多，却对程序运行时的行为产生重要影响。还有些源代码以外的内容，也可以影响到程序运行时的外观和行为。

资源文件是声音、图像、文本类的文件。程序在运行时读取它们，并把它们的内容表现出来。因此，如果资源文件的内容不同，程序的表现就会不一样，形成变体。比如，某个软件针对不同客户的变体，可能会使用该客户企业的徽标（Logo）。

有时候资源文件需要参与编译链接，但大部分情况下都不需要。

更广泛地讲，不同的数据也可以实现变体。与参数不同，这里的数据指的是大量的信息。

至此，读者可能会产生一个疑问，到底什么是变体？这确实是一个模糊的概念。一个程序，换一个背景图案或者来电铃声，算不算变体呢？好像不算。

一个程序，增加了三个用户定制的功能，算不算变体呢？应该算吧。那么，改变程序运行时的某个算法，比如从线性差值变为拉格朗日差值，算不算变体呢？

从程序的外观和行为，从程序使用者的角度，讨论是不是变体，就好像是讨论究竟有多少头发算秃顶，多少根头发不算秃顶。

而从软件开发的角度，问题的关键不是什么才算变体，而是具体通过什么手段，来实现程序外观和功能的不同。在本书中，我们把“程序外观和功能的不同”都归类为广义的变体，讨论实现的可能途径。至于你所在的开发团队是如何界定变体的，请和相关的同事讨论。

14.4　何时使用哪种方法

作为一般的原则，尽可能晚地实现变体。这样做，开发成本比较小。如果能在安装或运行时实现变体，就不要放到构建时，以避免编译众多的变体版本。而如果能在构建时产生变体，就不要使用多个分支，以避免分支间的合并，甚至是多个分支上工作。

这是一般的原则。还要考虑其他因素。比如：

- 如果某个独特的功能只是某个变体上具有的，其他变体都不需要。而且，在可预见的未来，也没有其他变体会需要这个功能。那么在这种情况下，用分支实现可能更合适。因为，如果把相应的代码放到主干上，就需要持久地维护它。时间长了，这类的独特功能多了，主干的包袱就会没有必要地变重。代码的结构也会变差，可读性也会变差。相对应的，设置的方法适合的是不同功能的组合，有些变体选择这些功能，有些变体选择另一些功能。比如，主干上有 10 个功能，变体 A 需要第 1、3、5 号功能，变体 B 需要 1、3、6 号功能，变体 C 需要 3、9 号功能……如果在构建时设置，那么每个功能对应一个编译开关。变体 A 就意味着简单地在最上层的 makefile 中打开第 1、3、5 号功能对

应的编译开关。

- 如果程序的运行效率，比如占用内存的量，运行速度等，是一个特别敏感的话题，那么要考虑构建时的设置，甚至是源代码分支，而非运行时的设置。以此尽可能地优化，提升效率。
- 如果有充沛的人力资源用于软件开发，而项目时间很紧，那么多考虑使用分支。这样，主干上标准版的开发和发布会更快，相应的变体分支上的开发会更快。当然代价是需要更多的人日。
- 如果程序开发所用的编程语言不支持构建时的设置，当然就不能使用这个方法。类似的，如果某个特性很难通过构建时或运行时的设置来实现，那么就不得不通过分支来实现。

14.5 设置对交迭的支持

设置这类方法不仅能够支持变体的开发，也可以在一定程度上支持交迭。

回顾一下我们为交迭开出分支的两个常见理由。第一，希望主干上开发出来的新功能，不要跑到准备对外发布的版本上去，不然公司就亏了。第二，希望主干上正在进行的新功能开发，不要影响发布分支上的代码的质量。

对于第一点，设置常常能够提供比较好的支持。新的特性用构建时的特定设置或运行时的特定设置才能激活。当然，可能因此程序会变复杂。

对于第二点，设置能够在一定程度上提供支持。但新的功能难免改到公共部分，难免需要对架构进行调整，因此常常有可能影响到即将发布版本的质量。

对于具体案例，如果第一个原因是主要的，第二个原因还不太重要，那么可以考虑在交迭之初，使用一段时间设置这类方法。待到接近发布，第一个原因变得很重要时，再加上分支这个方法。也就是说，用设置这类方法，可以延迟开分支的时间，尽管通常不能完全避免。

15

第 15 章　用组件的组合实现变体

除了分支和设置这两类方法外，还可以用组件组合这类方法实现变体。当你听到别人谈论软件复用这样的大词儿的时候，他们主要说的就是组件组合这类方法。

15.1 雏形：平台加应用

在第 13.1 节里，我们讲过变体产生的原因之一是客户定制类型的项目。为这类项目工作的软件开发团队经常会在做了 N 个项目后发现，其实这么多的项目，基础和核心的部分是相同的、可以共享的。而一些边边角角的东西不相同。比如：

- 图形界面显示的内容不相同，尽管背后的逻辑和算法都一样。
- 相关的硬件型号不一样，所以驱动程序不相同。
- 这个软件可以与另一类软件配合运行，但另一类软件有好几家公司都在做，至于版本就更多了。于是，与不同软件版本的接口就不同，对应于不同的连接程序。
- 有的项目里面，客户要求一些“额外的”功能模块。

这些边边角角的东西，映射到软件代码上，是一些周边的、外围的模块。而基础和核心的部分包括核心算法、基础架构等，是一大块代码，可以通用。通常，我们把这一大块代码称做平台。相应的，把周边的，外围的模块称做应用。把这种模式称做平台加应用。

其实不仅是客户定制类型的项目经常适合使用此模式。Internet Explorer、Firefox 等软件，在主程序之外还有很多插件，这些插件甚至是由不同公司或个人完成的。用户在使用时，可以自由选择喜欢的插件[1]。在这里，主程序是平台，那些插件是应用。

随着时间的推移，平台不断演进。1.0 版、2.0 版、3.0 版……平台是软件研

[1] 哦，有些插件不是用户选择的，是偷偷溜上去的……

发组织的核心资产、核心竞争力。软件开发组织应当投入足够多的人力物力，保证它的不断演进。而应用也很重要。应用是应不同客户、不同用户的需求而开发的，要尽可能满足他们的特定需求。顾客是上帝，只有让他们高兴，软件才能卖出钱。

从另一个角度看，在平台加应用这个模式里，程序是若干模块的组合。其中有一个特别大，被称做平台。其他的模块，被称做应用。这为我们提供了一个新的思路：程序，或者说是一个软件系统，可以是由若干模块，或者说是组件组成的。这个软件系统的变体，可以由不同的组件组成。而当某个组件出现在软件系统的不同的变体里的时候，可以是这个组件的不同的版本。这就是基于组件的软件复用，即本章的主题。

15.2　在不同阶段组装

系统由组件组装而成，组装可以发生在软件从源代码到运行中的程序这一生成转化过程中的不同阶段。

我们从源代码的组装谈起。源文件在一起构成目录，目录在一起构成更大的目录，依此类推，直到系统整体。或者从软件结构上说，文件构成模块，模块构成组件，组件构成子系统，子系统构成整个系统……那么从软件配置管理的角度，需要什么特别的支持吗？把它们一股脑放到一个版本库里，让版本控制工具管起来，不就行了？

如果在你所在的项目里，上述方法可行，那么你很幸运，不必再继续研究软件配置管理对源代码组件的组装的支持了。然而，在不少项目里，上述方法是不够的。原因之一是，对于大型系统的开发，其所对应的源代码量很大，开发人员又很多，如果用一个版本库，一些版本控制工具的性能会成问题。如果这样的问题确实发生了，或者明确地显露出这样的趋势，那么解决的方法之一是，把源代码分成不同组件，放到多个版本库中。这是一个工具层面上的原因，

这里不再展开讨论。

另一类原因是关于组件复用的。举例来说，如果把所有的源代码分成 1、2、3、4 共四个部分，那么产品 A 需要第 1、2、3 部分，产品 B 需要第 1、2、4 部分，产品 C 需要 1、3 部分。在这种情况下，从软件配置管理的角度，我们就会考虑把 1、2、3、4 各部分分别当做源代码组件。不同的产品实质上是本书中所说的广义的变体。也就是说，系统的不同的变体是由不同的源代码组件构成的。

即便是系统的不同的变体都使用了某个源代码组件，使用的内容也可以是不同的。也就是说，该源代码组件的不同版本，参与了系统的不同的变体的构成。

再次强调，我们这里所谈论的源代码组件，是从软件配置管理角度所划分的组件。划分关键是，一个组件是考虑软件复用时方便软件配置管理特别是版本控制的最小单位。如果从软件架构角度所说的若干个模块来看，它们永远是在一起，并且把它们作为整体来开分支、做基线就可以管理得很好，那就把它们当做一个源代码组件，而不是硬要分成多个组件。典型的例子是，在平台加应用这个模式中，平台尽管很大，但经常可以当作一个组件。

以上谈论的是源代码组件的组装。还记得我们在第 5.1 节谈到的从源代码到运行中的程序的全过程吗？大体上说，先是构建，然后是安装和运行。刚才讲的组件组装，发生在构建之前。组装还可以发生在构建之中，以及安装运行时。

发生在构建之中的组装，其典型的例子是静态链接库。静态链接库的常见形式是一些二进制的文件和一些文本的头文件。它们也存放在一个目录下。这个目录，可以认为是一个静态链接库组件。在构建的过程中，静态链接库组件参与进来，与其他源代码组件一起，生成可运行的程序，通常体现为安装包。

多种原因会导致使用静态链接库组件而非源代码组件。原因之一，如果静态链接库对应的源代码并非是本企业所研发的，而是另外一个企业研发并出售给该企业的，那么该企业可能不希望源代码外流，为此提供静态链接库组件而

非源代码组件。原因之二，在一个企业内部，如果某个研发团队负责某个公共模块的研发，这个团队希望对源代码有完全的控制，不希望其他团队在该模块上做任何修改，那么这个团队可能向其他团队提供静态链接库，屏蔽源代码。原因之三，如果某个模块很少改动，而构建时间又比较长，那么可以考虑做成静态链接库，从而在每次系统构建时，节约构建该模块的时间。

这些是使用静态链接库的好处。与源代码组件相比，静态链接库组件的主要弱点有两个。首先，如果有些变更需要同时修改库组件所对应的源码和源代码组件中的源码，就会比较麻烦。假定，库组件 A 与源代码组件 B 一起构成了一个系统。现在为修复一个缺陷，要同时修改 A、B 两个组件中的内容。库组件 A 是由源代码组件 A 生成的，必须首先修改源代码组件 A 中的内容。然后在对 A 组件自身进行集成时，生成库组件 A 的最新版本。这之后，在源代码组件 B 中，才能“看”到 A 组件的修改，进而修改 B 组件中相应的内容，这样就有了不少周折，耽误了时间。

在上面这句话中，“看”字加了引号，是因为 B 组件中的开发人员，并不能真正看到 A 组件对应的源代码，只能看到 A 组件的外在功能。这有些时候也给查找问题、编写代码、调试、测试等带来不便。这是第二个弱点。

组件组装不仅可以发生在构建之前，构建之中，也可以发生在安装运行时。比如，在 FireFox 上面安装一个插件，并不需要 FireFox 的源代码，也不需要构建 FireFox。需要的仅仅是安装这个插件，让它与 FireFox 的主程序在一起运行，在运行时相互配合。

很多软件在安装时提供一系列的选项，其中有一些选项是安装或者不安装某个功能模块，这也是安装运行时的组装。

在一些网站的网页上，你能看到一个 Google 搜索条，这个搜索条搜索的是本网站的内容。这就是 Google 提供的“Google 指定域名搜索”功能。在本质上，

Google 提供的这个功能成为了这个网站的一个组件。

在安装运行时发生组装，其原因与构建时组装的原因大体相同，也是上文那三条：防止源代码外流，防止随意修改源代码，节约构建时间。在最后一点上，甚至是省掉了构建时间。其弱点是不方便修改与调试，与构建时组装的弱点也类似，甚至更为严重。

什么时候适合源代码组件的组装，什么时候适合构建甚至是安装运行时的组装呢？以上三个原因都需要考虑。而对于弱点，一般来说，如果组件间相对独立，那么弱点就会小。而如果组件间耦合性比较强，那么麻烦就很大。从这一点来说，如果组件间相互独立，那么多考虑推迟组件间组装的时间。如果耦合性强，多考虑提前组件间组装的时间，甚至干脆就不要划分为不同组件来管理。

在第 14 章我们讲到，参数、资源、数据等非源代码的内容也可以支持实现变体。对于不同的变体，这些非源代码的内容可以是不一样的。因此，这些内容可以考虑作为单独的组件，供不同的变体使用。这样的组件可能在构建前参与组装，可能是构建中或安装运行时参与组装。

15.3 复合基线

我们仍然从源代码组件的组装说起。

还记得吗，在第 3.3 节里我们提到过，早期的源代码版本控制工具是以文件为单位进行管理的。每个文件有它的一个又一个版本，形成版本演进的历史。那么，如何记录一个程序的所有的源代码作为整体的版本呢？若干文件组成了产品，每个文件的特定版本组合在一起，是产品的特定版本。所以，记录产品的某个版本，只需要记录它是由哪些文件具体哪个版本组成的。这可以通过在各个文件上打标签等方法来具体实现（重画于图 15-1）。

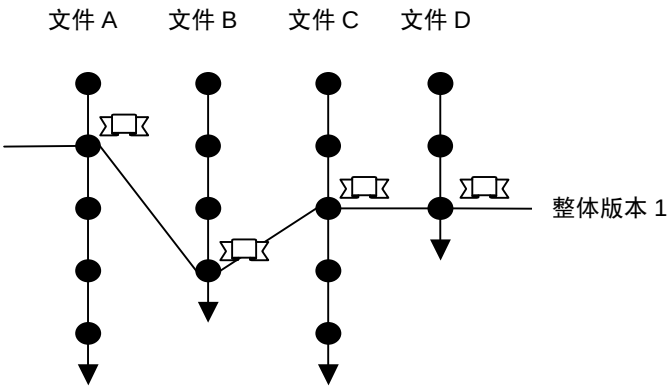


图 15-1　文件版本组成整体版本

当代源代码版本控制工具引入了变更集的概念，一个整体版本叠加上一个变更集形成另一个整体版本。这是使用当代版本控制工具时的主要视角。然而，整体的版本由部分的版本组合而成，这是一个具有普遍性的原理。在当代版本控制工具中，如果从这个视角看过去，该原理也是成立的，整体的版本是由各个文件的特定版本组成的，不管是不是用打标签这样的具体方法实现。

该原理具有普遍性。当源代码组件组装成系统时，我们通常直接应用该原理（见图 15-2）。

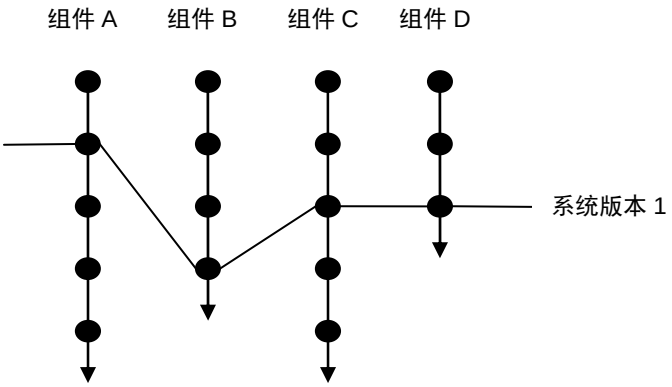


图 15-2　组件版本组成系统版本

通常，每个源代码组件放在一个版本库里。也就是说，一个系统的源代码存放在多个版本库里。如我们已经讲过的，对于每个源代码组件，每当有一个变更集提交的时候，其版本库就自然形成一个新的版本，该版本对应一个类似于 1、2、3……8573、8574、8575 或者 49A36B2、E331D55、8B52E13、94CC76F……这类的版本编号。

不同的组件的版本，组合在一起形成了系统的版本。系统的版本是一个客观存在，不论你是否记录它。如果某个时刻你灵机一动，打算记录它，那就拿出一张纸，把各个组件的版本编号抄下来，然后把纸锁到你的抽屉里。

当然我们有更好的方法，比如用一个文本文件来记录，然后把文本文件保存在安全的地方，甚至是版本库里。文本文件或许可以自动生成，而不用你把各个组件的版本号一个一个敲进去。至于各个组件的版本号呢，可以是看着头晕的自然编号，也可以是有意义的名称。而系统整体的版本号呢，名字是由你自己起的，当然最好是有意义的了。命名方法请参考第 13.6 节。

> **[Subversion]** 需要自己记录。考虑编写相应脚本。
>
> **[Git]** 如果你的源代码都放在 Git 版本库里，那么有个叫 Repo 的工具，能帮助你管理系统的版本。
>
> **[ClearCase UCM]** 直接提供了复合基线的功能，帮助你管理系统的版本。

这些是整个系统由源代码组件组装而成时的情形。当系统包括静态链接库组件时，或者系统由运行态的组件组成时，其原理也一样：系统版本由各个组件的版本来表达。只要记录了各个组件的版本，这个清单就代表了系统的版本。

原理都一样，实现方法可以不同。静态链接库以及运行态的组件是以二进制文件为主的组件，通常并不存放在源代码版本控制工具中。因此，不再是由源代码版本控制工具（或其相关的工具和脚本）来提供支持。对于二进制组件组成系统这种情况，除了手工完成或自己写些脚本来帮忙外，还可以考虑**软件包管理工具**或称软件包管理系统（Software Package Management System）。在第

18.1 节中，我们将介绍更多关于软件包管理工具的内容。

在第 4.3 节中，我们给出了基线的概念。一般来说，它是（狭义的）集成的成果，具备一定质量的整体版本。当系统由组件组成的时候，系统的基线由各个组件的版本，常常是各个组件的基线组成的。因此我们把系统整体的基线称做**复合基线**（Composite Baseline）。

注意，系统整体版本特别是复合基线的概念，并不总是那么重要。当系统的各个组成部分比较独立，当用户自己能够选择和更新各个组件时，整体版本的概念就不那么重要了。举例来说，Android 手机或者苹果手机，其软件由手机操作系统和在它之上运行的各个应用程序构成。用户可以自行下载或升级某个应用程序。用户会经常关心手机上这个软件系统的基线吗？当用户升级了某个应用程序之后，他会常想到要记录和保存这个系统的基线吗？用户更关心的是手机操作系统的版本和各个应用程序的版本，而非它们的整体版本。

15.4　复合分支

版本的概念既存在于各个组件，也存在于系统整体。组件的版本组成了系统的版本。类似的，分支的概念既存在于各个组件，也存在于系统整体。大致上，组件的分支组成了系统的分支。

下面是一个典型的例子。某系统由 1、2、3、4 四个源代码组件构成。每个组件都有其主干，开发人员把修改提交到主干上。每次集成工程师进行系统级集成，产生复合基线时，他选取各个组件的主干末端的版本，产生整体版本。

这是系统的标准版本。该软件还有一个名为 A 的变体，以满足 A 客户的一些特殊需求。对于 1、2 两个源代码组件，无须为 A 变体做任何修改。而对于 3、4 两个源代码组件，需要为 A 变体做一些修改。此外，还有第 5 个源代码组件，其内容是专为 A 变体而存在的。

开发团队经讨论，决定变体 A 采用这样的分支策略：在 3、4 两个源代码组件上，为变体 A 开出分支，用于集成变体 A 相应的特殊修改。在 1、2 两个源代码组件上，不必为变体 A 开出相应分支。在第 5 个源代码组件上，主干就是只为变体 A 服务的。

有变体分支，就有合并。假定该开发团队中，这是由集成工程师主导完成的工作。每当系统标准版有新的复合基线的时候，把该复合基线上的内容同步到变体 A 上。如何完成呢？具体地说，对于 3、4 两个源代码组件，集成工程师用 A 分支上的最新内容填充他的工作区，然后把系统标准版的复合基线所对应的该组件的版本合并过来，联系开发人员解决出现的任何合并冲突。对于 1、2 两个源代码组件，集成工程师用系统标准版的最新复合基线所对应的该组件的版本来填充他的工作区。对于第 5 个组件，集成工程师用主干上的最新内容填充他的工作区。现在，在集成工程师的工作区中的源代码，是把系统标准版的新的复合基线同步到变体 A 之后的结果。接着，在工作区中，编译构建、粗略测试，如果都没问题，就提交工作区中 3、4 两个源代码组件上合并的结果到公共版本库[2]，从而完成从标准版的最新复合基线到变体的系统级同步。

当然，除了同步之外，集成工程师还要做该变体的（狭义）集成工作。对于 1、2 两个源代码组件，他选取最近一次从标准版到该变体的系统级同步时所确定的，在主干上的版本；在 3、4 两个源代码组件上，他选取 A 分支末端的内容；在第 5 个组件上，他选取主干末端的内容。如果编译构建、粗略测试都没问题，就产生该变体的新的系统级基线。[3]

那么，为变体 A 修改代码的开发人员，他们如何工作呢？每次更新工作区时，对于 1、2 两个源代码组件，他选取最近一次从标准版到该变体的系统级同步时所确定的，在主干上的版本；在 3、4 两个源代码组件上，他选取 A 分支末

[2] 并且以某种方式告诉在变体 A 上工作的开发人员，更新工作区时，对于 1、2 两个组件，可以使用新的版本了。请参见随后描述开发人员更新工作区的段落。

[3] 细心的读者可以看出，如果把每次从系统标准版合并到变体 A 的工作，与一次（狭义）集成的工作一起做，就能省下一次构建和粗略测试的时间。

端的内容，并与本地修改（如果有的话）合并；在第 5 个组件上，他选取主干末端的内容，并与本地修改（如果有的话）合并。

为变体 A 修改代码的开发人员，只应该修改 3、4、5 组件的内容。每次提交时，把代码改动提交到对应的分支。对于 3、4，是 A 分支。对于 5，是主干。

在这个典型例子中，存在着两条系统级分支。一条是系统标准版的，一条是变体 A 的系统级分支。为系统标准版工作的开发人员，从该系统级分支所对应的各组件的分支上更新工作区，并提交代码到这些组件级分支上。集成工程师为系统标准版做集成时，取对应的各组件的分支上的最新内容，生成系统级基线。

变体 A 的系统级分支并不意味着它对应于每个版本库中的一条名为 A 的组件级分支。对于 3、4 组件，它的确对应于相应版本库中的名为 A 的组件级分支。然而，对于第 5 个组件，在本例中，它对应的是该组件的主干。对于第 1、2 个组件也是。

变体 A 的系统级分支也不意味着它可以修改每一个组件所对应的分支。在本例中，第 1、2 个组件对于变体 A 来说是只读的。

系统级分支的末端也不一定是各个组件所对应的分支的末端并随其增长。在本例中，当开发人员从变体 A 的系统级分支的末端更新代码到本地时，对于 1、2 组件，取得的是集成工程师最近一次从系统标准版更新到变体 A 时，所采用的该组件在主干上的一个版本，该版本属于当时系统标准版的最新复合基线。也就是说，在变体 A 上工作的开发人员，他用系统级分支的末端来更新自己的工作区时，对于 1、2 组件，他并非取得某个组件级分支的末端。

同样，当集成工程师为变体 A 做集成，生成基线时，对于 1、2 组件，取得的是集成工程师最近一次把系统标准版同步到变体 A 时，所采用的该组件在主干上的一个版本，该版本属于当时系统标准版的复合基线。也就是说，当集成

工程师取得系统级分支的末端时，对于 1、2 组件，他并非取得某个组件级分支的末端。

当某条系统级分支的末端在某个组件上并不对应于某条组件级分支的末端并随其增长时，我们也可以认为，在这个组件上，这条系统级分支对应的是一个组件版本，而非组件分支。

尽管组件级分支的名字可能与系统级不同，尽管组件级分支不一定像系统级分支那样可以由这个变体修改，尽管系统级分支的末端也不一定是各个组件所对应的分支的末端，这里还是有一些共同点。那就是，系统级分支由组件级分支构成。在这种情况下，我们把系统级分支称做复合分支。

> **[Subversion]** 需要自己记录。考虑编写相应脚本。
>
> **[Git]** 有个叫 Repo 的工具，能帮助你管理系统级的分支。
>
> **[ClearCase UCM]** 直接提供了流的功能，系统级的流由组件级的流或基线组成。有一个限制，组件级的流必须与系统级的流同名。

以上讨论的是系统由源代码组件组装而成的情形。当系统包括静态链接库组件时，或者系统由运行态的组件组成时，其原理也一样：系统的分支由各个组件的分支组成。只要描述了各个组件的分支，描述了是否对于该变体只读，描述了总是对应于组件分支上最新版本还是某个特定版本，这个清单就代表了系统级分支。

原理都一样，实现方法不同。静态链接库以及运行态的组件是以二进制文件为主的组件，通常并不存放在源代码版本控制工具中。因此，不再是由源代码版本控制工具（或其相关的工具和脚本）来提供支持。对于二进制组件组成系统这种情况，除了手工完成或自己写些脚本来帮忙外，还可以考虑软件包管理工具。没错，软件包管理工具不仅可以帮忙管理复合基线（见上节），也可以帮忙管理复合分支。如我们已提到的，在第 18.1 节中，我们将介绍更多关于软件包管理工具的内容。

对于二进制组件，通常不能直接做分支间的合并。合并工作需要由相应的开发团队在源代码形式下完成。另一方面，对于并非采用源代码版本控制工具管理的二进制组件，其分支的表现形式也有所不同。分支通常由版本号的命名来体现。比如，1.3.2.114、1.3.2.115、1.3.2.116 就是在一个概念上的分支上。如果系统级分支定义该组件取 1.3.2 系列的最新版本，那就相当于说，取 1.3.2 这个概念上的分支的末端，也就是 1.3.2.116，如果还没有 1.3.2.117 的话。

就像复合基线一样，复合分支，并不总是那么重要。当系统的各个组成部分比较独立，当用户自己能够选择和更新各个组件时，复合分支的概念就不那么重要了。

15.5　实现软件复用：分支、设置、组件相结合

软件复用（Software Reuse）又称代码复用（Code Reuse），试图通过利用已有的软件，更快更省力地生产出新的软件。从这个意义上说，不论是基于 1.0 版开发 2.0 版这样的历史演进，还是基于标准版产生变体版本，它们都属于软件复用。

事实上，演进和变体这两个概念，时常是混在一起的。与 1.0 标准版很相似的变体 A，跟与 2.0 标准版很相似的变体 B，它们之间是变体关系，还是演进的关系？两者都有。

不仅是演进、变体这两种情况属于软件复用的应用范畴。在硬件领域，我们常能见到一些通用零件，比如一个螺丝钉，这个型号的螺丝钉可以用在摩托车上，也可以用在缝纫机上。摩托车和缝纫机之间，是演进的关系吗？显然不是。是变体的关系吗？显然也不是。其实在软件领域也类似。如果你用 Visual C++ 编程的话，大概很熟悉 MFC[4]。在 MFC 的帮助下开发的软件，它们之间也即非

[4] Microsoft Foundation Classes，简称 MFC，是一个微软公司提供的类库（Class Libraries），以 C++ 类的形式封装了 Windows 的 API，并且包含一个应用程序框架，以减少应用程序开发人员的工作量。

演进关系，也非变体关系。不论是硬件领域的那颗螺丝钉，还是软件领域的 MFC，在本质上都是一个通用组件，可以用来组成各种各样的系统。这当然也是软件复用。

如何实现软件复用？我们已经讲解了三类基本方法：第 12 章和第 13 章中的分支，第 14 章中的设置，以及本章中的组件组装。尽管在讲解设置和组件组装这两类方法时，我们主要讲解了如何用它们来支持变体，但是显然它们也支持其他情况的软件复用。

分支、设置和组件组装这三类软件复用的基本方法，经常联用。在第 14.5 节，我们给出了分支与设置联用，支持历史版本的例子。而上一节，给出的是组件组装与分支联用的例子：组件组装产生不同的系统时，选取某个组件的不同版本，常常是选取该组件的不同分支上的版本。也就是说，在组件级，使用了分支来支持复用。而在系统级，同样可以用分支来实现复用。这时的分支是系统级的复合分支。

15.6 实现软件复用：架构、过程和组织

实现软件复用，常常不是一件简单的事。实现软件复用，需要软件配置管理的支持，但绝非做好软件配置管理即可。它需要一个好的、适合复用的软件系统架构。它需要一个好的、适合复用的研发过程。它需要一个好的、适合复用的管理结构和人员组织结构。

- **一个适合复用的软件系统架构**

如果打算使用组件组装方法，那么首先应该是一个组件化的软件系统架构。组件内部，高度聚合。组件向外提供明确清晰的接口。组件之间，通过接口，相互配合。

这里所说的组件，可不是一个类或一个函数级别上的。一个系统，可能有成千上万个函数和类。直接管理它们，仍然是困难的。这里所说的组件，是粒度比较大的组件。它可能是若干基础函数和类的集合、一个工具箱，供应给程序使用。它可能是某个基础设施，是程序运行的框架和中枢神经系统。它也可能是实现了系统对外的某个或某类功能。这些组件，可以体现为一个目录下的全部源代码，可以体现为一个静态链接库，也可以体现为一个运行的程序或运行时调用的动态链接库。

使用面向对象语言编程，有利于实现组件化及组件复用。但这并不是一个充要条件，既不充分，也不必要，仅仅是提供了一定的方便，提供了一定的促进作用。相比之下，面向对象设计和面向对象分析更为重要。

关于软件系统架构，除了要组件化，另一个重要方面是，要层次化。要根据通用程度，使用分层体系结构来组织软件系统。一个典型的分层系统是这样的：

最底层是系统软件，这是指操作系统，以及一些对特定硬件的访问的封装。

第二层是中间件。这一层，是为了提供独立于操作系统的一层平台和环境。它提供一些通用的工具，更为重要的是，提供一些分布式对象计算的基础性支持。比如 J2EE 就属于这类平台和环境。通用的数据库及其接口也在这一层。

第三层是与特定业务有关的层。这些是与该软件开发企业所涉及的特定业务领域有关的工具和平台。这一层，在一个企业里，被广泛地共享使用。

第四层，也就是顶层，是应用系统层。这一层是用户看到的，在其上直接操作的应用软件系统，是软件开发企业向外提供的产品。

- **一个适合复用的研发过程**

以产品为中心的软件研发过程，是不适合复用的。这样的软件研发过程，会集中精力于一个又一个产品本身，而很难顾及到公共的部分，顾及到资产的复用。

在第 13.4 节我们提到，要把变体间公共的改动放到主干上，放到标准版本上。这个标准版本，是公共部分，是主流。要集中力量于主流的演进。

当谈到软件复用的时候，公共部分，是一系列的公共组件，是其中每个组件的一个或几个常用版本。几个常用版本，可能是组件演进中产生的历史版本，也可能是组件的变体版本，或者两者都有。

在构成应用系统的时候，除了用到公共组件的常用版本外，可能需要开发新的组件，也可能需要改造常用版本，以适应应用系统的需要。

引入了软件复用的研发，应该以这一系列公共组件的常用版本的研发为主要关注点。关注它们的开发，关注它们的版本演进。因为对它们的投资，能够在很多个应用软件系统中得到回报，很值。

而对这一系列公共组件的常用版本的研发，又不能脱离具体的应用。否则，就可能造出用不上的组件、用不上的功能。

在一系列应用系统的总体规划阶段，就应该仔细考虑整体软件架构，考虑开发哪些公共组件，开发公共组件的哪些常用版本。还要考虑，每个应用软件系统使用哪些公共组件的哪些常用版本，需不需要改造，需不需要开发新的组件。接下来，在一系列系统的漫长的研发过程中，需要不断细化当初的规划，也需要监视并根据实际情况调整当初的规划。

在研发过程中，要保持公共组件的常用版本的公共性。对于比较常用的功能，应该尽量在公共组件的常用版本中实现，以利于共享，降低开发成本。另一方面，很特殊的功能、很特殊的要求不应该放在这里完成，可以考虑组件变体的方法，或者采用新的组件。

而在具体的源代码编写、集成和测试工作中，需要提供良好的支持，使得公共组件的常用版本能融合到具体的应用系统中，变得有用，得到验证，避免

偏离。同时，又要保证公共组件的常用版本的研发，具有一定的独立性，不要受到具体应用系统研发的过多干扰。

- **一个适合复用的人员组织结构**

在研发过程中，考虑对系统标准版本，以及对公共组件的常用版本的研发，单独立项，保证其具有一定的独立性，保证有足够的资源供应。这里的资源供应，主要指的就是人力资源。应该有单独的项目组，来开发公共组件的常用版本。

而对于每个具体的应用软件系统，一般来说，也应该单独立项，以保证有足够的资源，用于这个特定的应用软件系统的开发，以保证这个将来要销售的产品，其研发工作有人管，有人盯着。

而在这两者之间，应该有人进行总体规划工作和随后的协调工作。这些工作，有些是偏技术的，比如软件系统总体架构的确定等；有些是偏管理的，比如制定某个公共组件的常用版本的发布和使用计划。这些工作，应该由另一组人员来承担，以便独立于具体组件的研发，也独立于具体应用软件产品的研发。

软件复用绝不仅仅是软件配置管理的事情，尽管软件配置管理在其中扮演了重要的支持性角色。实现复用不是这么几段文字就能够说清楚的，需要扩展成一整本书。限于本书篇幅，这里仅在本节做概要介绍。

16

第 16 章　支持多地点开发

“他们说，来吧，我们要建造一座城和一座塔，塔顶通天，为要传扬我们的名，免得我们分散在全地上。”[1]神害怕这座塔真的盖起来，于是就把人们分散到了全世界不同的地方，并且让他们讲不同的语言。但是，即便有如此不利的条件，人们仍在努力沟通，相互协作，共同工作。不过，这回不是盖塔，而是开发软件。

[1] 《圣经 · 旧约 · 创世纪》第 11 章第 4 节。第 11 章第 1 至 9 节，是关于巴别塔的故事。

16.1　何时出现多地点开发

大型跨国企业，其软件研发也常常是跨国的。不同国家、不同地点的团队共同工作，共同完成软件系统的开发。不仅是在大型跨国企业，在一些国内企业中，这种情况也同样存在，可能是在不同的城市，分别成立研发站点，共同开发。这样做的原因之一是，在一个国家、一个城市，招不到足够多的合适的软件研发人员。而大型软件系统的研发需要上千人甚至上万人的共同努力[2]。这样，就构成了多地点开发。

不一定非要是大型软件系统的研发才需要多地点开发。有些时候，一些软件项目需要现场工作。于是派遣一个软件开发团队到达现场，进行一段时间的工作。在这段时间里，这个漂泊在外的团队与大本营之间就构成了多地点开发。不一定是派遣了一个团队，可能只是派遣了一两名工程师，在现场做调试和少量修改。在这种情况下，该工程师与大本营之间同样构成了多地点开发。

据说有些人不喜欢办公室的环境，喜欢在家编程序，或者找个山青水秀之地，一边听鸟叫一边编程序，而团队其他的成员仍在办公室里。当然，也有可能所有的团队成员都不在办公室里。比如对于开源软件的研发，通常就没有这么个办公室，全靠大家业余在家完成。这样的情况也构成了多地点开发。

第三方软件组件的开发，通常是人家在人家自己的地方开发出来的。这也构成了多地点开发。

[2] 对于跨国企业，这样做的原因之二是，利用时差，由不同国家的员工轮班倒着干。北京这边，提交系统测试请求后，天黑了，回家睡觉。此时，芝加哥那边，天刚亮，收到请求，立即开始测试，并在天黑前完成工作，报告给北京。北京这边，第二天一早，就看到了测试结果。这样做的原因之三是，不同地区的人力资源价格相差很大，能力也有所不同。通过寻求全球范围的最佳资源配置来节约研发成本、提升研发效率。

16.2 主要困难和解决方法

在不同地点进行的共同开发，与在同一地点开发相比，会遇到很多额外的挑战。

在不同地点开发，形成了某种隔阂。打电话沟通、电子邮件沟通，总没有面对面沟通方便、容易、有效率，于是人们倾向于不做这样的沟通。不光是在普通开发者之间，存在沟通这个问题；开发者和管理者之间，也有这个问题；管理者和管理者之间，同样有这个问题。于是，就会倾向于自成体系，自己确定各类事情。这可能使得不同地点开发的内容接口不一致，无法集成在一起；也可能使得不同地点开发的内容，外观风格不一致，看起来像两个产品拼凑而成，等等。

对此起到推波助澜作用的还有时差。德国慕尼黑时间中午 12 点，为解决某个着急的技术问题，一封电子邮件从慕尼黑发往中国北京。北京这边是晚上 19 点，员工已经下班了。等 14 个小时后，在北京时间早上 9 点，信从北京发出，飞向美国亚特兰大——因为这个问题是亚特兰大那边的一个专家最熟悉的——这时候，是亚特兰大当地时间晚上 8 点，员工下班了。再等 13 个小时后，在亚特兰大时间早上 9 点，信件从亚特兰大发出，飞向慕尼黑。这时候，是慕尼黑时间下午 3 点。在慕尼黑的员工，在敲了 27 个小时的手指头后，终于收到了回复[3]。

多地点开发可能会引入不同的文化，这也会造成一些困难。典型的是母语不同，使用英语不熟练。还有可能存在思维方式上的不同，管理理念上的不同，等等。

以上这些都是在人的层面上。在环境和工具的层面上，同样面临挑战。例如，在版本控制方面，两个地方的工程师要修改同一处源代码，如何控制？让

[3] 信上就两个字：挠挠。

他们远程访问一个版本库，还是让他们在当地各有一个版本库？那么这两个版本库之间如何协调？在缺陷跟踪方面，在一个地点录入的缺陷记录，如何让另一个地点的开发人员看到，并且在修改源代码后，改变该缺陷记录的状态？需求管理工具、项目管理工具等，也同样存在多地点开发的挑战。

面临这些多地点开发的挑战，该如何应对？

首先，尽量避免多地点开发，除非有充足的商业理由。

其次，如果一定要采取多地点开发，那么，可以让每一地点的团队或个人，负责相对独立的一块源代码或完成相对独立的工作任务。比如，以组件为单位，把不同的组件，分配给在地理上不同地点的团队开发。而组件间的接口要清晰简单，并且应该尽早讨论、确定并记录。这样，有助于减少对不同地点间人员沟通的需求，也减少对环境和工具的要求与依赖。

再次，要加强协调和配合。要有适当的人员进行总体的管理，进行各站点之间的沟通和协调。要有合适的流程保证各站点之间的同步。比如，两个站点之间开发内容的集成，要预先确定集成的频度，并依此执行。

最后，环境和工具本身应该尽量支持多地点开发。包括一般环境的支持，比如多点电话会议，甚至视频会议；也包括软件管理各领域的环境的支持，比如软件配置管理方面——这是下面几节讨论的内容。

16.3　离线工作模式

假定这样一个场景，张大侠偶尔在家工作一天，因为要等人来修空调。为此，在前一天下班前，他把整个产品的源代码下载到自己的笔记本电脑上，保存到自己的工作区里，准备回家后，修改其中的一些内容。第二天，当他回到公司后，与公司内部网络连接上，执行一些操作，让版本控制工具自动识别出，在昨天离线工作的时候，哪些源代码被改动了。然后，把这些改动算作一个变

更集，提交到公共版本库里。

> **[Subversion]** 支持这样的操作。
>
> **[Git]** 当然支持这样的操作，因为连私有版本库都带回家了。
>
> **[ClearCase UCM]** 支持这样的操作。ClearCase 会检查出离线工作期间，所有被改动的文件，把它们称做是被劫持的（Hijacked）文件。注意，要支持离线工作，必须使用快照视图，不能使用动态视图。因为使用动态视图时，源代码实际是在服务器上，而不是在本机上。

采用这样的方法有一个要求，就是开发人员要能够把整个产品的源代码，至少是自己关心的那些部分的源代码，下载到自己的笔记本电脑上。这要求代码量不是太大，公司内网的网速不是太低，自己的笔记本电脑上还有足够的磁盘空间。这样的要求，如果在办公室里工作，可能是不需要的，因为自己的工作区、工作环境可能是在某台服务器上；执行各种操作，本质上也是在服务器上执行。而自己的笔记本或个人台式机，仅仅是用于登录，仅仅作为接收键盘鼠标输入，并把相关内容显示到输出的终端。

若能满足上述要求，一般来说，如果是开发人员偶尔在家工作一两天，那这个法子不错。时间再长，可能就会有问题了。原因之一是，在离线状态下，当要完成多项工作任务时，没有办法区分哪些代码修改属于哪项工作任务，只能把这些代码修改全算在一起，作为一个变更集；原因之二是，在离线状态下，无法从服务器上获得更新，不能更新工作区到其他开发人员的最新提交或集成工程师标出的最新基线。

这样的讨论，同样适合于个别开发人员出差的情况。如果出差期间，只是进行少量修改，那么离线工作没有问题；如果修改量比较大，应该对应于多个变更集，或者出差期间需要拿到公司总部那边的最新源代码，那就不合适了。

如何区别记录多个变更集？解决的方法之一是，在工具不能支持的情况下，自己动手，丰衣足食。比如，为了区分多个变更集，可以对每个变更集完成后的整个产品源代码断面进行一次保存、备份。这样，离线期间完成了几个变更

集，就产生了几个产品源代码整体断面。在回到公司连上内网后，在版本库里建一条分支，导入离线期间生成的那几个源代码整体断面，然后再合并回主干。

解决的方法之二是，改用更好的工具：分布式版本控制工具。这样，就可以在离线状态下，在本机私有版本库中保存多个变更集。

> **[Subversion]** 集中式版本控制工具，不支持这样的操作。
>
> **[Git]** 分布式配置管理工具，支持这样的操作。
>
> **[ClearCase UCM]** 集中式版本控制工具，不支持这样的操作。

解决的方法之三是，远程访问公司的公共版本库。这是下一节的内容。

16.4　远程访问模式

如果开发人员在外的时候，能够访问公司内部的资源，包括版本库、源代码等，那就不仅解决了区分记录多个变更集的问题，也解决了与服务器上公共版本库同步的问题，甚至还能提交自己的修改到公共版本库。这是总的方法。至于具体实现，可以有以下途径。

其一，开发人员把自己的工作区、工作环境，放在公司的自己的电脑或者某台服务器上。当开发人员不在公司的时候，通过拨号、广域网、虚拟专用网等技术[4]，连接到公司这台机器，在这台机器上登录，执行各种操作。而自己手头的笔记本或个人台式机，仅仅是与这台机器连接的终端。

这个方法的好处是，对网络速度的要求不太高，特别是当采用命令形式的字符终端界面，而非图形终端界面时。这个方法的弱点是，所有的实在的东西，包括源代码和可运行程序等都在公司里，不在开发人员手头。如果客户当时需要什么，需要额外的网络传输。

[4] 严格地讲，拨号、广域网、虚拟专用网并不是并列的关系。详情请咨询公司 IT 部门。

其二，开发人员通过拨号、广域网、虚拟专用网等技术，连接到内部网，就好像在公司里一样操作，以完成各种工作。包括，用公共版本库中最新的源代码更新工作区，修改源代码，提交变更集等。

这个方法的好处是，一旦连接到内网，开发人员的工作方法和在办公室里时一样，不需要额外培训和设置。而弱点是，网络连接速度比较慢，使得各类操作的速度比较慢。

一个小技巧：可以在离开公司前，先下载产品源代码到自己笔记本上的工作区里，以免远程访问时，为此耗费太多的时间。

其三，使用与在内网环境中工作时不同的连接方式。比如，在内网环境中，使用某个版本控制工具特定的网络协议，以获得较高速度。而在远程连接时，使用 HTTP 网络协议，遵守这样的协议的软件，在经过授权后，可以访问公司的版本库。遵守这样的协议的软件，可能是一个专用的客户端软件，也可能就是像 Internet Explorer 这样的通用的网络浏览器。当然，用通用的网络浏览器来访问，功能可能会受限。比如，可能只能看，不能改。

能够这样访问，首先要求公司对外提供这样的服务。这需要相应的软件，也可能是版本控制工具自身能够提供这样的功能。

> **[Subversion]** 通过配置，Subversion 服务器端可以对外提供基于 HTTP 等传输协议的通信方式。因此，可以通过 Internet Explorer 这样的通用的网络浏览器访问公共版本库，进行查看。而如果使用 Subversion 客户端，则可以远程完成全部日常操作。
>
> **[Git]** 在服务器端，把 Git 和 Web 服务器相连接，就可以对外提供基于 SSH、HTTP 等传输协议的通信方式。如果在服务器端安装了 GitWeb 等软件，就可以通过 Internet Explorer 这样的通用的网络浏览器访问公共版本库，进行查看。而如果在客户端使用 Git 本身，则可以远程完成全部日常操作。
>
> **[ClearCase]** ClearCase 远程客户端（ClearCase Remote Client，CCRC）提供了这样的远程访问功能。前提是，在公司内运行着 ClearCase Web 服务器，并且外界能访问。

值得一提的是，如果用后两种访问方法作为解决方案，那么在实际操作时，会发现使用分布式版本控制工具（详见第 3.5 节）会比使用传统的集中式版本控制工具快很多。这是因为，分布式版本控制工具回避了大部分远程访问操作，访问本地的版本库就行了。

上述三种远程访问方法，都要注意安全性的问题，要防止源代码等泄露给竞争对手。事实上，只要是多地点开发，这一点都要注意。

16.5　当地版本库模式

当谈到访问速度的时候，远程访问再怎么好，也没有当地访问好。远程访问通过广域网，这通常达不到局域网的速度。能不能变远程访问为当地访问呢？能不能把版本库拿到当地呢？如果是一票人马都飘泊在外，聚集在一处临时的或永久的开发地点，那么，这种需要会变得更加强烈。“凭什么公司总部那边可以方便快速地访问，而我们这边，只能慢吞吞地远程访问？要知道，我们这边的工作，主要是自己配合自己，我们主要关心邻桌写的源代码，要与邻桌写的源代码集成，却要去公司总部那边兜个圈……”

一些版本控制工具提供了不同地点的多个当地版本库相互配合的功能。这主要有两类方法。

第一类方法，每个研发地点的工程师都可以访问并且只访问该站点当地的公共版本库。也就是说，当地的公共版本库就能够支撑起当地的研发环境。当然，当地研发和其他地点的研发，需要协调配合，相互同步。因此，各个站点的公共版本库之间需要进行数据交换。版本库之间的这种通信，可以是通过通信网络进行的，也可以是通过快递一张光盘来实现的。这在技术上都是可行的，因为一般来讲，当地开发人员对当地的公共版本库进行操作时，不需要当地的公共版本库立刻与其他站点的公共版本库打交道[5]，也就是说，不需要当地公共

[5] 以便节约时间，并且回避网络不稳定的风险。这正是使用当地版本库带来的好处。

版本库与远方公共版本库之间的实时通信。

在这种情况下，不同站点的开发人员，如果都向所在当地的公共版本库中的同一条分支上提交，就会产生版本混乱，因为同一条分支在不同的站点有了不同的内容。对此基本的解决办法是，使用不同的分支。在每个公共版本库中，只能把代码改动提交到属于该站点的分支上去，让这条分支不断向前演进，而不能把代码改动提交到属于其他站点的分支上去。也就是说，那些不属于这个站点的分支，尽管可能存在于这个站点的公共版本库中，但对于这个站点是只读的。

不同地点的开发人员，工作在不同公共版本库的不同分支上。而这些分支，需要适时合并以便同步。本质上是不同地点的开发人员的工作需要集成在一起。具体的合并方法可以是：在主控站点，各分支上的内容被合并到了主干；而在其他站点，从主干合并到当地分支。

以上是不同地点的公共版本库相互配合的第一类方法：当地的开发人员只访问当地的公共版本库。另一类方法是，同时访问当地的公共版本库和主控站点的公共版本库。当地的公共版本库和主控站点的公共版本库之间，不再是双向同步，而是从主控站点的公共版本库到当地站点的公共版本库的单向同步。当地的公共版本库供当地开发人员下载源代码，初始化或更新自己的工作区[6]。而当开发人员完成工作，提交修改时，直接上传到主控站点的公共版本库，而非当地的公共版本库。

这样做的理由是，从性能方面考虑，下载的网络流量远比上传的网络流量大，因此宜从当地公共版本库而非位于另一个站点的公共版本库下载，或许能把下载时间从 5 小时缩短到 1 小时。而上传到哪个公共版本库，对性能的影响不大，或许只是 1 秒钟和 5 秒钟的区别。而从功能方面考虑，如果各个站点的

[6] 如果使用的是分布式版本控制工具的话，那么每个开发人员从当地的公共版本库初始化或更新自己的私有版本库，然后才是工作区。

修改都提交到主控站点公共版本库的同一个分支上，那么此后就不用再做合并工作。因此，从当地公共版本库下载，但自己的代码改动要上传到主控站点的公共版本库。

最后，注意区分一下概念。本节讲的是多个站点相互协作时，每个站点都有一个当地的公共版本库。这些公共版本库之间会进行某种同步。而我们在第 3.5 节里介绍分布式版本控制工具时，重点讲的是每个开发人员在自己的机器上都有一个私有版本库。这个私有版本库会与该站点的服务器上的公共版本库相互同步。

当然，分布式版本控制工具不仅能够支持一个站点内的公共版本库与私有版本库之间的同步，也自然而然地能够支持多个站点的公共版本库之间的同步。另一方面，不仅是分布式版本控制工具能够支持多个站点的公共版本库之间的同步，一些集中式版本控制工具在一些附加工具的支持下，也能够进行多个站点的公共版本库之间的同步。

> **[Subversion]** 可以考虑使用 SVK、Subversion MultiSite 等第三方工具来实现上述第一类方法，进行不同站点的 Subversion 版本库的同步。第二类方法靠 Subversion 本身的版本库同步和备份功能就能实现。
>
> **[Git]** Git 是分布式版本控制工具，对此提供内在的支持。两种方法都可以实现。
>
> **[ClearCase]** ClearCase Multisite 是 ClearCase 的一个附加工具，它支持一个版本库的多个副本之间的同步，即第一类方法。

16.6　只读模式

以上三节，我们一直在谈不同地点的开发人员，需要修改同一块源代码。但这并没有涵盖多地点开发的所有情况。比如，有公共组件 A、B、C，在公司总部开发，而在异地开发的某项目，会用到公共组件 B、C，同时自己开发组件 D。管理的策略是，在该项目中，不对组件 B、C 做任何修改，仅仅是在需要时从公司总部拿到新版本。

除拿到只读不改的源代码组件外，类似地，还有可能拿到静态链接库组件、运行组件。总之，都是只读的。也就是说，一个地点是生产者，而另一个地点是消费者，仅仅是使用。这样，就可避免两地团队同时修改一处源代码的问题。

对于这样的场景，就不一定需要版本控制工具本身对多地点开发的支持了。可以采用其他的方法来传送只读组件，比如 FTP 或刻成光盘快递。但是要注意，作为软件配置管理，无论是否使用工具，都要保证系统的正确配置，即使用哪些组件的哪个版本。

16.7 多地点开发中的变更管理

变更管理特别是变更请求管理，同样需要支持多地点开发。因为一个变更请求，比如缺陷，可能是在甲地发现并提交到变更请求管理工具，在乙地通过评估，在丙地被修复，在丁地被验证。一条缺陷记录，可能需要在不同的地点被操控，更别说被查阅了。

实现这一点的基本方式，是实现变更请求管理工具的网上操作。即通过通用的网络浏览器或特定的界面，访问变更请求管理工具。这既包括浏览，也包括创建新的变更请求条目或修改已有的条目。

把变更请求管理工具放到网上，要比把版本控制工具放到网上来得容易，而且使用起来也方便，速度也快。这是因为，变更请求管理工具仅仅是记录变更请求信息。而版本控制工具不仅要记录版本信息，控制版本变更，还要存储并向外提供源代码本身。

> **[Bugzilla 等开源工具]** 它们本身就是基于 Web 的。
>
> **[ClearQuest]** 提供了通过 Web 访问的方式。

也可以考虑把变更请求工具背后的数据库分布在全球各地，每个地方有一个本地副本。这可以加快访问速度，并且有可能对变更请求管理工具进行一些

本地的设置和定制。一些变更请求管理工具支持这样的方式。为防止冲突和混乱，每条变更请求记录只属于某一个站点，其他站点只能浏览，不能修改。如果确实要修改，比如 B 站点要修改属于 A 站点的一条变更请求记录，那就先要把这条变更请求记录的主控权从 A 站点转移到 B 站点。这可就涉及 A 站点和 B 站点的两个变更请求数据库之间的通信了。因此有点麻烦，也可能会有延迟。

> **[ClearQuest]** 提供了对这种分布式的方法的支持。

17

第 17 章　支持跨组织合作

据说全世界的 MBA 们都在读《世界是平的》[1]。越来越多的商品，从设计到制造，是由不同国家不同企业共同完成的。在软件领域也是这样。

[1] 一个美国人，托马斯·弗里德曼（Thomas L. Friedman）写的畅销书。他在书中讨论全球化。

17.1　从开源到外包

老板找到张大侠，让张大侠写一个图像转换程序，把 BMP 格式的图像成批地转换成 JPG 格式的，同时在图像右下角贴上公司的徽标。下周一交给他。哦，太难了，难道要为此研究 JPG 格式的压缩算法？听说那个算法是很复杂的。张大侠哭丧着脸，从老板办公室里出来。看来这周末得加班了。张大侠确实有点一根筋[2]，是不是？

王大侠看不过去，给张大侠支了一招。网上有把 BMP 转换成 JPG 的程序，而且是开放源代码的。只要基于这个程序，稍微改动两处：写个关于贴徽标的函数，在转换过程的适当位置调用；把整个转换过程封装，自动地反复执行。张大侠照此办理，添添改改了 50 行源代码，很快就交活了。据说为此获得了老板的表扬[3]。

是的，网上有大量的开放源代码可用，而且不少质量还不错。为什么不用它们呢？不少免费的程序，随时可以下载安装使用，为什么不用它们呢？

从获取资产的来源上分，除了开放源代码和免费产品这类外，还有两种途径。一种是其他厂商大量出售的标准软件，比如操作系统；一种是其他厂商专门生产的软件，这常被称为外包。这三种途径都可以获取软件资产。这样获取到的资产，称做**第三方资产**（Third Party Asset）。第三方也被称做供应商（Vendor），因此对它的管理称做**供应商管理**（Vendor Management）。

这些第三方资产，其形态可能是源代码，可能是静态链接库，也可能是可以直接安装运行的程序。

---

[2] 我们在第 2.2 节里提到过，一根筋的张大侠。
[3] 口头表扬而已。还用说。

从使用的角度来看，这些第三方资产可能被加以改造，就像上一个例子中，这常常是发生在取得的是源代码的情况下。它们也可能被当作组件，集成进一个大的系统。

这些第三方资产可能还不断有后续的版本，由原来的厂商提供。

17.2 管理第三方资产

从软件配置管理的角度对这些第三方资产的管理，主要有以下几点：首先，要适当地标识和记录它们。不仅是要记录资产的名称，还要记录资产的版本。除此之外，还应适当记录资产来源方面的信息，这样便于日后更新版本，也便于万一弄丢了补回来。这些信息，可能是通过存放目录的名称、基线名称、分支名称来反映的，也可能是有个文本文件或表格，登记它们。

其次，要存放好这些资产，并能方便地访问和使用。可以把它们放进版本库，也可以放进共享目录，还可以只是简单地放在每个开发人员的电脑里。这要看具体情况。一个拿过来的开源软件的源代码，应该放在版本库里，特别是如果还要对它加以改造的话。而买来的 Windows 操作系统，就得放在每个开发人员的电脑里了。

再次，如果使用了第三方资产，应该对此进行记录。在整个系统中，如果使用了第三方组件，那么应该清楚地反映出，用的是哪家的，什么组件、什么版本，以便于将来复现。如果对外来资产进行了改造，那么需要清楚地反映出，当初是基于哪家的，什么产品、什么版本。这些信息可能已经通过存放目录的名称、基线名称或分支名称反映出来了，也可能由文本文件或表格对它们进行记录。

最后，要管理外来资产的版本变更。这些变更，可能是由外界完成的，外界产生了新的版本，需要拿过来。当然，既要及时地拿过来，跟上时代，也要

小心，不要拿尚不成熟的新版本，干扰了工作。外来资产的变更，也有可能是因为自己在上面加以改造，产生了新的版本，这种情况常发生于外来资产是源代码的时候。而如果外来资产的变更在这两方面都有，那就比较有挑战性了。

在本章中，会详述如何应对这一挑战。不过，在这之前，我们先讨论一下容易一些的、但与之相关的话题，这是下一节的内容。

17.3　本地记录第三方源代码的演进

在理想的情况下，外部使用者可以随时查阅取用存储在供应商的版本库里的源代码，多方便！然而，即便处于这样的理想情况下，使用者也仍会考虑把这些源代码在自己的公司里存一份。因为东西不在自己手里，就是让人不放心。会不会网络连不上了？会不会他们被黑客攻击了？会不会他们不善管理，把版本库弄得一团糟？会不会将来有一天，他们把开源的源代码藏起来，改成收费的？这些都是问题。总之，通常使用者是要把这些源代码拿过来，在自己的公司存一份的。具体的说，是导入到本公司的版本库里。

在本公司的版本库里，也希望在一定程度上记录第三方对其源代码的修改，就好像记录本公司开发人员对本公司源代码的修改一样。如果第三方和本公司使用的是同一种版本控制工具，那么可以考虑把第三方的版本库中的内容精确地复制过来。

这是理想情况。如果两边用的不是一种版本控制工具，或者如果要是人家不乐意提供这些信息，只愿意对外提供发布版本的源代码断面的话，这个方法就行不通了。那就这样，把发布的整体版本拿过来，假装每个新的发布版本的产生，都是由于在已有的发布版本上，完成了一个大的变更集。

具体的操作方法是这样的。导入第三方源代码当前的整体版本到自己的版本库里合适的位置，从此开始管理。当第三方提供了源代码的新的整体版本的

时候，先是签出自己的版本库里存放的第三方源代码到工作区，然后用新的源代码覆盖它。在版本控制工具看来，这个覆盖，就好像是开发人员手工修改源代码一样。这之后，签入并提交到版本库。这样，就在版本库中产生了源代码的新的整体版本。在版本库中，这个新的整体版本与旧的整体版本相比，就是多了一个变更集。而在这个变更集里，记录了两个整体版本间全部的差异（见图 17-1）。

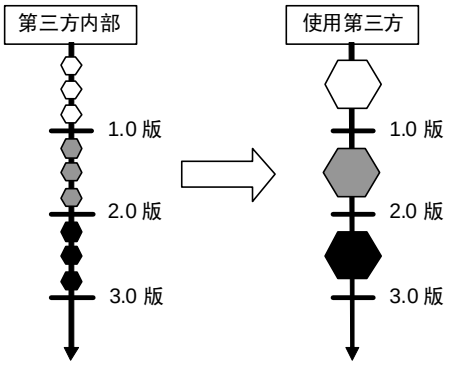


图 17-1　记录第三方的演进

如果再讲细一点，那就是，在工作区中，当用新的源代码覆盖旧的源代码的时候，要注意，不光是一些已有的文件被其新的版本覆盖，还有可能是有新的文件产生。而一些旧的文件，则需要删除。还有一些旧文件，被改了名字后，出现在新的整体版本中。而目录也有可能变化，可能有的目录没有了，有的目录是新添的，还有的是改了名字。对于这些特殊情况，需要特殊处理。可以用手工的方法通知版本控制工具，哪些要删除、添加或重命名。也有可能版本控制工具自身提供了某种程度上的自动化的方法。

> **[Subversion]** 自带一个脚本，可以方便地不断导入第三方源代码的新的版本。
>
> **[Git]** Git 甚至支持持续地从另一个版本控制工具（比如 Subversion）中导入，导入完整的历史记录信息。

这样，在自己的版本库里，就会有一个又一个新的整体版本，这些对应于

第三方提供的一个又一个版本。当然，也可能并不是第三方提供的每个新版本，都在自己的版本库里可找到。可能我们嫌麻烦，隔几个版本才导入一次。

17.4　再加上本地的改造

在本章开头的例子里，张大侠不仅使用了第三方的源代码，而且对其进行了改造。在导入了第三方的源代码到自己的版本库后，张大侠对其进行了修改，最终产生了老板想要的东西。事实上，张大侠是在主干上直接完成了这一改造，并且在主干上发布了自己的作品。

可是，有的时候，我们既要进行本地对第三方源代码的改造，也要跟随第三方源代码的标准版本演进。那怎么办呢？哦，我们有分支这个法宝。

一边是本地进行的改造，一边是标准版本的演进，两条路，谁做主干？从技术上讲，都可以。因为，主干的本质就是分支，只不过是特殊的分支。从标准版本的视角看，标准版本是主流，应该占据主干，而本地改造后的版本，本质上是变体，应该在另一条分支上。而从本地使用的视角看，本地使用的版本，是最应该关注的，是主流，它应该在主干上。第三方不断提供的标准版本不过是“那边”又给过来的一些更新而已，应该用另一条分支把这些更新记录下来，并最终引入到主流里。

既然两种方法都可以，并且没有本质的不同，那么，我们在这里把它们命名为改造分支和标准分支。分别对应于主干和另一条分支，或者对应于另一条分支和主干。

本地进行的改造与标准版本的演进，两方面的修改要合并在一起。如何合并？肯定不能是把改造分支上的变更集复制到标准分支上。因为，每次标准分支根据第三方提供的新版本向前演进的时候，是不管三七二十一完全覆盖的，所以曾经从改造分支上复制过来的变更集的内容，会被完全覆盖而丢失。

可以把标准分支上的变更集复制到改造分支。标准分支上的每一个变更集，代表着前后两个标准版本之间的差异。不断地把这些差异拿过来，也就使得改造分支的内容“跟上了形势”，随标准版本的演进，而向前演进。注意，这并不意味着，一旦标准分支有所前进，就需要立刻把新的内容拿到改造分支上，可能不需要这么着急。可以在标准分支向前长了几个标准版本之后，再把它上面相应的变更集一股脑地复制过来。

还有一个办法，可以在标准分支的最前端，再建一条新的改造分支。然后把原改造分支上的内容，复制到新的改造分支上。这样，新的改造分支上既有原改造分支上的改动，也就是本地改造的内容，又基于最新标准版本，所以达到了两方面合并的效果。这是用“换底”的方法，让改造分支“跟上形势”。这个办法，适用于不需要频繁地“跟上形势”的情况，否则要创建很多条改造分支。这个办法适用于本地改造的内容不多，而第三方的工作做了很多的情况。这样，在变更集复制时，这个方法与前一个方法相比，容易对引入的代码变动进行检查和审核。

是不是听起来有点耳熟？没错，在讲如何用分支处理变体的时候，我们也给出了类似的两种方法，并且进行了类似的优劣比较。这是因为，在本质上，第三方标准版本和本地改造版本的关系，就是标准版和变体的关系（见第 13 章）。

在讲变体的时候我们谈到，有时候在变体的开发上，也会原发地进行公共的改动，这些改动应该复制到标准版里。在标准版由第三方提供的情况下，也应该尽力做到这一点。如果可以访问并提交修改到第三方的版本库，那就可以考虑在第三方的版本库里直接完成，这可能是把本地完成的修改，重新做一遍。如果不能访问第三方的版本库，就把相关变更信息，比如缺陷现象、缺陷分析、缺陷修正方法的描述及相关源代码，描述并打包，通过电子邮件等方法，提供给第三方。做这些事情有两个理由。第一，有利于减少本地版与标准版间的差异，使得本地版将来不断维护，不断提升版本的工作变得容易些。第二，把本

地的工作成果通过标准版与全世界分享。第二个理由也是个很重要的理由，因为涉及道德情操。

17.5　共同开发

本章前几节一直在讲一件事，就是如何使用第三方提供的内容。而如果站在第三方的角度看，是如何把内容提供给需求方。这是一个对外发布软件的过程，只不过发布的内容除了二进制形式外，还有可能是源代码形式，都不太难。

困难的是另外一种情形：与另外一个组织共同开发。这种情形与第 16 章讨论的情形比较像，但是更困难。很多原因，文化上的、流程上的、商务上的、政治上的……我们这里还是更关心软件配置管理方面的原因。不同的开发组织可能采用不同的工具来支持开发。比如，版本控制工具，甲公司用的是 ClearCase，而乙公司用的是 Subversion。它们之间的协调同步要比使用相同工具时的协调同步更困难。另一个重要原因是，一个开发组织可能不希望外界直接看到其内部使用的版本库，更别说直接把修改提交到该版本库上了。

作为一个基本的策略，尽可能划分好各个开发组织的职责，落实到源代码组件上。尽可能避免两个开发组织同时修改同一处代码。比如，组件 A 由甲公司开发，乙公司只是使用其静态链接库。这样，就与源代码控制工具无关。

而如果不方便使用二进制的静态链接库或可执行程序，那就由甲公司向乙公司提供组件 A 的源代码。乙公司把源代码导入乙公司内部的版本库，不论其版本控制工具是否与甲公司相同。

当这一点都难以做到，即乙公司需要修改甲公司提供的组件 A 的源代码时，有两种情况。一种是，乙公司所做的修改，其中的一部分值得送回甲公司。对于这样的情况，请详见第 17.4 节中关于在第三方源代码上加上本地改造的讨论。

另一种情况是，乙公司所做的修改全部需要送回甲公司，因为两个公司是

在为某个项目、某个产品共同修改这个源代码组件。对于这样的情况，请参见第 17.4 节对于当地版本库模式的讨论。注意，由于两个公司使用的版本控制工具不同，两个版本库之间的同步，需要参考本章第 16.5 节讲到的本地记录第三方源代码时使用的方法。另一方面，如果开发人员提交代码修改到另一个公司，由于版本控制工具不同以及权限的限制，需要一些额外的操作，比如通过电子邮件把相关信息传送给另一个公司的开发人员，让他代为提交。这类似于上一节中的讨论。

在两个开发组织共同开发的时候，除了源代码版本控制，变更管理也会变得比较麻烦。其原因类似：不同的变更管理工具，权限的约束……其解决策略和解决方法也类似。

18

第 18 章　管理软件部署

我们一直在关心软件研发过程中的软件配置管理。而软件研发的目的是为了将来能使用它。为此需要把它部署到运行环境中，让它运转起来。这可能很简单，把安装包发给客户，让他自己装一下就行了。但有时候就比较复杂，假如有数十个甚至数百个彼此相关的程序一起运行在数十台甚至数百台服务器上。

18.1 安装包的存储和管理

我们在第 5.1 节讲过，通过一系列的生成转换过程，源代码变成正在运行的程序。典型的过程：编译、链接、生成安装包；安装、运行。前三步常被统称为软件构建，而后两步则属于软件部署。后者是本章的主题。我们从安装包的存储和管理讲起。在第 5.5，我们讲过一些最基础的内容，本节将介绍更多。

不仅在研发过程中，在软件研发企业内部，安装包是需要存储和管理的。对于最终用户来说，安装包同样需要管理。用户经常希望能够方便地从网上下载到他想要的安装包，因此需要以某种形式提供安装包的存储和下载的对外服务。即便没有这么方便，安装包提供者也应该有个合适的地方存储安装包。当有客户需要某个安装包的某个版本时，能够迅速通过互联网之外的某种方式把安装包提供给客户。

不论是对内还是对外，各类安装包应该有相对统一的存放地点。简单的方法是使用一个共享目录，并设置了合适的读写权限。注意，如果安装包的种类比较多，那么应该对共享目录内目录的结构有一个规划，把安装包适当分类，按类别划定子目录。并且，日后要持续维护这个结构。

不仅要存储安装包的最新版本，还要适当存储历史上的版本。因为有的时候要退回到历史上的版本去，要把历史上的版本与最新版本进行比较。

安装包通常本身已经携带了关于自己的一些信息。比如，安装包的名称、产品名称、产品版本号等。安装包内自带的说明文件，有更进一步的说明。而安装包存放的位置，也常常暗示了产品名称、所属分类等信息。如果这些信息仍然不够，或者是不够规范，那么考虑表格、数据库等形式，记录各安装包的详细信息。例如安装包来源方面的信息，以及这个安装包版本的质量等级。后

者对于自产自用的软件特别重要，因为在这种情况下，开发过程中尚不成熟的安装包版本，与可以发布给客户或部署上线的安装包版本，有可能是放在一起的，需要用质量等级标志来区分它们。

除了记录每个安装包的信息，还应该适当地记录安装包之间的关系，这主要是运行时的相关关系。在第 15 章讲解基于组件组装的软件复用时，我们提到，可运行的组件在一起组装成可运行的系统。这些可运行的组件经常是以安装包的形式出现的。因此，安装包之间具有一定的依赖关系。准确地讲，是不同安装包的各自版本之间或版本系列之间，具有依赖关系。比如，产品 A 是一个数据库产品，可以独立运行；而产品 B 是一个工作流产品，它的运行，依赖于产品 A。而产品 B 的 1 系列和 2 系列版本，只能运行于产品 A 的 1 系列版本上。如果产品 A 升级到 2 系列版本，产品 B 就不能运行，除非产品 B 升级到它自己的 3 系列版本。

当安装包的种类特别多，相互关系十分复杂的时候，用手工记录、手工检查的方法可能就不合适了，需要借助管理工具。运用我们在第 15.3 节提到的软件包管理工具，管理安装包及它们之间关系的信息，也管理安装包的存储。事实上，本章下面几节将谈到的内容，同样应当考虑运用工具来管理——当管理对象比较多，情况比较复杂的时候。

> **[RPM]** RPM，全称 RPM Package Manager，是 Linux 世界里使用最广泛的软件包管理系统。类似的还有 Debian 上的 dpkg 等。
>
> **[PyPI]** PyPI，全称 Python Package Index，是用 Python 语言写成的软件的官方存储库。它与 CPAN 有点像。CPAN 存储的是 Perl 语言写成的软件。
>
> **[Android Market]** Android Market 是一个由 Google 为 Android 系统用户创建的服务，允许安装了 Android 系统的手机和平板电脑用户从 Android Market 浏览和下载一些应用程序。用户可以购买或免费试用这些应用程序。苹果公司的 App store 也类似。

18.2 管理安装过程

在 MS Windows 上，安装一个软件，一般来讲挺容易。安装包通常是一个可执行程序，或者安装包包含一个可执行程序。运行它，会出现一个图形界面的安装向导，它会带领你一步一步地进行安装，包括进行一些设置，最终安装好。

> **[Windows Installer 和 InstallShield]** Windows Installer 是 MS Windows 上最流行的安装工具，能安装以 msi 为扩展名的安装包。然而制作这样的安装包可不是一件容易的事，幸亏有了 InstallShield 来帮忙。
>
> **[Inno Setup 和 NSIS]** 在 MS Windows 上，如果你想免费制作安装包的话，可以考虑上述两个开源软件。

在 UNIX/Linux 系统上，有时情况会有所不同。可能拿到的安装包里面是源代码，需要你在本地编译链接它，才能生成可执行程序。可能你需要自己把这些可执行程序等挪到合适的地方；而在进行设置的时候，可能也不是通过图形界面，而是修改某些文本文件。

> **[RPM]** RPM 让 Linux 世界里的安装工作简单多了。

这些都是常识性的东西。那么，除此之外，是什么因素让问题变复杂了呢？

其一，人们担心如果安装出错，会带来不利的影响。如果只是给个人安装个 mp3 播放器，那当然无所谓了。但是如果是给银行 ATM 机升级软件，那可就不得了了。试想一下，忽然之间 ATM 机前人群喧闹，因为不插卡它也吐钞票。

安装出错，可能是因为安装程序本身的质量，也可能是因为拿错了版本；可能是因为安装程序与环境不匹配，也可能是因为在错误的时间进行了安装，等等。

为此，安装也需要做计划。何时安装、安装哪些内容、怎么安装、有什么前提条件。应该考虑到如果安装出现问题，能否回退到安装前的状态，如何回退。应该考虑到先做一些试验性安装，没问题再正式安装。安装应该得到适当

的批准。在具体安装前，应该仔细进行检查，检查版本、检查运行环境、检查相关关系等。在安装时，要严格按照预定的步骤安装，并观察是否有异样。要记录整个安装的情况，以便将来回顾。在安装后，应该做一些粗略测试，以验证安装成功。

其二，人们希望系统在安装期间，特别是在系统升级的时候，仍平稳运行，向外提供服务。至少是安装一个系统的时候，不要影响其他系统的运行。如果能做得更好，则希望这个系统本身在不知不觉中升级。

实现这个目标的一种方法是，在原来系统仍在运行的情况下，安装、设置和调试新系统。如果没问题，就切换过去。比如，目前有 5 台服务器，向外提供搜索服务。可以先停掉其中的一台，在这台服务器上，升级搜索服务程序的版本；之后测试，确认无误后，这台服务器上线；再停掉下一台。依此类推。

其三，人们希望能够批量地安装，在不止一台机器上安装。还以上文的搜索服务为例，假如不是 5 台服务器，而是 500 台[1]，那么，总不能一台一台人工安装吧。需要有某种自动方法，能够成批地安装。

为此，首先需要能够远程安装，而非人工登录每一台服务器；其次，每台服务器上的安装过程、安装位置和安装设置等应该相同。这样有利于自动化，也有利于管理；再次，每台服务器的安装过程，应该尽可能地实现自动化，不需要人工干预。这些条件都能达到后，就可考虑自动成批安装了。

另一种方法是，复制整个服务器的环境到其他服务器上。这样，只需要一开始安装好一台服务器，然后成批复制就可以了。

[1] 这个数不是说多了，而是说少了。不论是 Google 还是 Yahoo，搜索服务器的数量都比这个多。不过，具体是多少台，我可不能告诉你。

18.3 记录运行系统配置

在研发过程中，源代码存在着配置。不同的源代码组件配置成源代码系统，或者说是构成了一个产品的源代码整体。在版本库中，可能存在着多个产品的源代码。除了源代码之外，在研发过程中，还存在着静态链接库、文档等，这些都是软件资产，都是研发过程中软件配置管理的对象，要管理它们的配置。

而当程序在用户环境或者在测试环境运行起来之后，同样需要关心运行时的配置。具体地讲，我们要关心，当前运行着哪些系统、每个运行系统的版本、每个系统有哪些运行组件、每个运行组件的版本是多少。每个系统、每个运行组件，是运行于什么硬件上，直白地说，是运行于哪台服务器上，运行于什么样的服务器上。

就像研发过程中的配置管理一样，对线上运行也需要把这些配置识别出来。也就是说，标识出并记录下这些运行系统和运行组件，他们的名称和版本号；记录下运行系统和运行组件的安装设置、参数设置；标识出并记录下各个服务器的名称和硬件情况。除了记录下这些条目外，还要记录这些条目的关系：运行系统由运行组件构成；运行系统间可能相互影响；运行组件和运行系统物理地存在于服务器上。

这些信息用于回答一系列基本问题：装了什么？装在哪儿了？装的是哪个版本？

这些信息，如果信息量不大，情况不太复杂，用简单的文本记录下来，就基本可以了。甚至可能不用记录，因为太简单，谁都知道，或者操作系统本身提供了查阅安装了哪些产品的功能[2]，或者产品本身提供了查阅其名称和版本号的功能，需要时查看一下就可以了。而如果信息量比较大，可以考虑借助于表

[2] 例如，MS Windows 操作系统提供了这样的功能。如果安装的每一个产品或组件，都向操作系统进行了注册的话，通过操作系统提供的“添加或删除程序”这个工具，可以看到所有已安装在本机的程序。

格和数据库。而使用专门的管理工具，可能很方便。

除了当时的配置信息外，历史上的配置信息也同样需要考虑记录。需要记录历史上某个时间点，特别是一些指定的关键的时间点当时的系统配置。这样，有利于将来恢复到历史上某个点去。此外，还需要记录每一次实际发生的变更，比如：何时，哪个系统或组件，发生了什么变更。这有利于在出问题后进行分析，是什么原因造成的问题，进而考虑如何恢复。

18.4　管理变更请求

在系统运行过程中，可能会有来自各方面的人，想对正在运行的系统进行某种调整。可能是要求修复缺陷，可能是要求增强已有功能，可能是要求增加新功能，可能是要求提高运行性能。这些请求都可以看做是对运行系统的变更请求。对这些变更请求需要进行管理，就好像在软件研发中，需要对变更请求进行管理一样。

事实上，这些变更请求的一部分就是对软件研发的变更请求。但是，并非所有这些变更请求，都需要送到软件研发部门。有些变更请求，是对当前运行系统配置的变更请求，也就是说，是对运行哪些系统，包含哪些组件，使用哪些服务器等方面的变更请求，也包括对运行系统的一些设置、使用的一些数据的变更请求。

这些请求，就像软件研发的变更请求一样，需要记录并跟踪，以防丢失。需要分析评估，考查它的成本，它的风险，它所带来的收益，综合各方面情况，经过适当讨论和批准，决定是否要实现这个变更。如果确实要实现这个变更，就小心谨慎地实施并验证。比如，如果是安装或升级软件，那就要参考本章第 18.2 节里描述的那样进行。

在实施变更的前后，要通知相关受影响的人。比如，如果某个银行的软件

系统升级，需要停止服务 4 小时，那可是件很大的事情，需要使用各种方法，尽可能地通知到所有相关的用户，为此提前做好准备，错开这段停机时间。而如果没有广泛通知，就会被当作一次事故，人们还可能会猜测，有更大的问题隐瞒起来了，备不住还会引起挤兑风潮。

对于运行系统的变更请求的管理，同样不可以教条。如果要对全公司的 MS Windows 操作系统装 Service Pack，那是件需要正规管理的事情。而如果是某个用户想把他本机的桌面背景换一下，恐怕就不用请示谁了。

在这里，我们融会贯通一下。还记得我们提到过环境管理么？包括管理编译环境、运行环境和开发环境。其实，这些环境本身也是运行系统。所以，进行软件研发的环境管理时，可以参考本章内容。

现代企业的运行，大都需要软件系统及其背后的硬件的支撑。对软件运行系统及其背后的硬件的管理，统称为 **IT 管理**。IT 管理，包括了软硬件的问题管理、变更管理、配置管理、发布管理、容量管理等。IT 管理有一个流行的模型叫 ITIL（IT Infrastructure Library）。另一个 IT 管理架构是 CobiT（Control Objectives for Information and Related Technology）。

18.5 管理网站

这是一个危险的话题。说它危险，首先在于它的内容很丰富，也很有意思，因此搞不好就会写出十几万字，占到本书篇幅的一半，导致吓跑一半以上的潜在读者；说它危险，还在于它的概念、涵盖面和方法，远还没有形成比较一致的观点。

从软件配置管理的角度看网站。首先，它是运行系统。它可能包含不止一个运行系统，而每个运行系统，包含很多个运行组件。因此，从运行系统的角度对其进行配置管理，可以参考本章前面几节的内容。

运行系统、运行组件是由源代码系统、源代码组件生成的。这里，可能有交迭，可能有变体，可能有与第三方的合作。从软件研发的角度对其进行配置管理，可以参考本书中若干相关章节的内容。

网站的软件，与一般软件有所不同。一般软件通常需要编译链接才能运行，而网站软件通常是用脚本语言写成的，不需要编译链接这些步骤，只需要打包并安装。这个对软件开发和管理的影响不大，不构成困难和挑战。

然而，网站开发和管理的真正困难在于，有时候不好分辨，哪些算是软件研发的输出，哪些算是运行时的配置和设置，哪些算是系统运行行为本身。比如，如果想在页面上再加一个新闻链接，这算不算软件研发？要不要走软件研发的变更请求和版本控制？或者，这算不算是对运行时配置和设置的修改？要不要走运行系统的变更请求和配置管理？再或者，这只是网站运行时的一般性操作，根本不需要跟研发部门和 IT 部门打交道？

因为不好分辨，所以更要明确责任。要明确，哪些内容由谁来负责，如何修改。而它的前提是，网站有比较好的系统架构，并且有比较好的工具支持，能够把脚本程序的变化、页面编辑产生的变化和运行时的变化，三者分离开来。

分析上面这个例子，假如，这些新闻是存储在数据库里的，上面所述的页面，是根据数据库内容自动生成的，那么，加一个新闻链接这件事，只需要在数据库里增加相应内容，这就跟研发部门和 IT 部门没关系。而如果这个新闻链接是在网页里写死的，那就需要网页编辑改动该页面，这时就要对线上内容进行版本控制，防止因为多个人同时修改，而使改动丢失。假若与此同时，研发部门正在对该网页甚至该网站的格局进行大的调整，那么还要想办法，保证将来上线的新系统包含了在研发期间对线上内容进行的修改。这就比较有难度了。

因此，网站要有比较好的系统架构，并且有比较好的工具支持。其中的工具主要指网站内容管理系统（Content Management System，CMS）。当前，网站

内容管理系统的内容涵盖模糊不清，各有各的说法，各有各的实现。这里不做深入介绍。

网站内容管理系统，通常也包含了一些版本控制甚至变更管理的功能。也有的时候，是网站内容管理系统调用了版本控制工具。还有的时候，是在网站内容管理系统的外部加上版本控制，管理其输入或输出的内容的版本。

网站内容管理系统大有发展空间，让我们拭目以待。

19

第 19 章　软件配置管理实施

本章是本书的最后一章。在本章，我们简略介绍一下实施软件配置管理的一些要点。

19.1 不同阶段，不同挑战

第一阶段，一个软件开发组织初创之时。这时候的当务之急是，建立起软件配置管理的基本解决方案。其中最重要的是，引入源代码版本控制工具，把源代码管理好，防止源代码丢失、版本覆盖、版本混乱等各种问题。有了源代码版本控制工具，开发就可以比较顺利地进行了。随后的要紧事，是建立起基本的缺陷跟踪系统，其主要目的是让测试团队和开发团队能够有效地合作。

第二阶段，软件开发团队的规模不断增长之时。这通常伴随着源代码量的迅速增加，编译构建所需时间的不断延长等。这一系列的变化给集成（广义和狭义）带来了越来越大的压力。在这个阶段，我们的关注重点是围绕集成的优化。比如，如何在狭义集成前、集成中、集成后合理地分配质量保证工作？如何让构建更快？如何让测试更快？能不能让狭义集成过程完全自动化？如何实现持续集成？等等。

第三阶段，发布多个版本、多个项目、多个产品之时。此时的主要挑战是软件配置管理对软件复用的支持。实现软件复用有多种模式、多种方法。注意选取适当的模式，并注意不同模式同时使用时，如何相互配合。这些模式中，以组件的组合对软件配置管理的挑战最大。另一方面，当版本、项目、产品的数量很多时，数量本身也带来挑战。比如说，三五年前的某个项目，你知道它的源代码就在公司的某个版本库的某个目录下，但是到底是哪个版本库哪个目录呢？

以上三个阶段的划分，只是常见的、典型的情况。并且阶段与阶段之间，也并没有非黑即白的明确的界限。请以此为参考，根据你所在的研发组织的实际情况，确定当前的工作重点。

19.2　如何完成一项改进

不论是引入一个版本控制工具，还是改变提交源代码修改的步骤，抑或是调整关于内部版本命名的规则，在本质上，都是对现有的软件配置管理解决方案进行一项改进。这一项改进可能是关于软件配置管理工具的，或软件配置管理相关流程的，或两者都有。不论完成一项什么样的改进，都有一些通用的步骤和要点。

在改进方案调研和选型时，要对所在研发组织的实际需求有真切的了解。比如，现在亟待解决的问题是什么，对某项指标有多高的需求。举例来说，某个工具号称能支持 2000 人同时使用，而你所在的研发组织，现在只有 20 人，未来三年的目标是发展到 100 人，那么这个工具的这个高指标对你所在的研发组织并无意义。工具软件厂商和服务咨询的提供方可能会夸大你的需求，暗示你有某方面的需求，以此来引诱你花大价钱购买其实你并不真正需要的工具或服务。

用免费的工具还是收费的工具？传统上认为，收费工具通常质量更好，出了问题能得到及时的修复，关于工具如何使用，也有工具厂商能够提供专业的解答。然而，相应的开源免费工具，如果足够流行，被成千上万的软件研发组织使用，那么它的质量通常已经相当好。并且，有些新兴的开源工具与一些有年头的收费工具相比，明显具有更好的架构，因此有更好的性能，也更容易维护和使用。在今天，开源免费的工具，有可能是一个更合适你所在的研发组织的选择。

某个改进，特别是流程上的改进，在研发组织作为整体因此而受益的同时，可能也有一些部门、一些团队、一些个人没有因此得到好处，甚至受到损害，比如工作步骤变得比以前繁琐。他们可能会成为变革的阻力。如何解决？方法

之一是，识别出这个改进的主要受益方，与相对应的管理者沟通，请求他的帮助。他经常有很大的热情，从管理者的角度去协调和推动这项变革发生。

从改进方案的讨论阶段开始，就要注意保持与利益相关方（Stakeholders）的沟通。确定改进方案确实是他们想要的，尽量满足不同方面的需求。在确定方案后，如果实施改进所需的时间较长，那么应该适时汇报阶段性的成果。

尽可能把你心中的宏图伟业，落实为一个个小的改进项目或阶段。每一个都有明确的产出，对研发组织有实质性的贡献。小的改进与大的改进相比，风险较小、实现较快、可根据反馈及时调整，比较容易得到批准、付诸实施。并且，通过一个个成功的改进，不断累积你的信用，日后打算进行一些革命性的改变时，会更容易获得支持。

一项改进本身的质量保证工作该怎么做？除了改进需求评审、方案设计评审、代码评审、在测试环境测试等方法外，**试点**（Pilot）也是一种广泛使用的方法。试点是在真实的环境中小范围地试用新的方法。这样，出了问题负面影响也不至于很大。同时，由于是在真实的环境中试用，如果没什么问题，那么就可以比较放心地推广。

要成功实施一项改进，培训、支持、文档也很重要。通过培训等方法，让具体的使用者充分了解为什么要改变，改变到具体什么样的流程和操作方法。当使用者遇到问题时，要提供方便的途径，让他们的问题得到解答或解决。培训和支持都是向使用者提供的服务。如果能够让使用者在很多时候自己帮助自己，那就更好了。如何做到呢？如果有合适的相关文档，文档具有合适的结构和合适的内容，易于使用者阅读、便于使用者查找，那么就能够在相当程度上减少支持使用者所需的人力。

19.3　在一个项目的生命周期中

项目生命周期中的软件配置管理流程如图 19-1 所示。

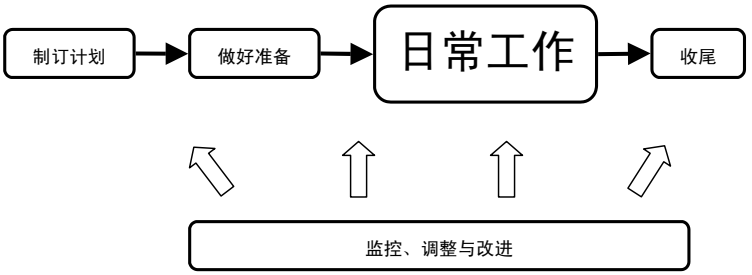


图 19-1　项目生命周期中的软件配置管理

- **制订计划**

凡事预则立，不预则废[1]。制订计划，意味着考虑在这个产品的开发过程中，软件配置管理要怎么做，这包括软件配置管理的方方面面。

这说起来很大，其实做起来并不难。通常在做这个项目时，会在相当程度上遵循和使用该企业已有的软件配置管理解决方案。只是需要确定一下，这个项目放在版本库的什么位置，在缺陷跟踪工具里叫什么名字之类的问题。

当然，具体的项目，可能会有它的一些特殊之处，需要仔细地研究，确定相应的解决方案。比如，如果这个项目是公司至今为止第一个使用某个供应商提供的源代码组件的项目，那么就需要仔细研究，以确定分支的命名、多久做一次导入、如何送回公共改动等一系列问题。类似的还有，开发团队是否分布在不同的地理位置，不同的产品之间是否存在复用，等等。这些是这个项目中软件配置管理的特定**场景**（Scenario）。

[1] 出自《礼记·中庸》。原文为“凡事豫则立，不豫则废；言前定，则不跲；事前定，则不困；行前定，则不疚；道前定，则不穷。”

对项目的软件配置管理进行计划，并不意味着一定要写一份名叫**软件配置管理计划**（SCM Plan）的厚厚的文档。作为计划的成果，有些内容可能写在了某个 Wiki 页里，另一些内容填在某个表格里，甚至有些内容干脆就是写在一封电子邮件里告诉大家。

- **做好准备**

要做哪些准备工作，深受组织级软件配置管理工作的影响。

如果公司已经提供了可用、适用的软件配置管理工具，如版本控制工具等，则只需简单地为这个产品进行初始化工作。比如，在其中创建项目、创建目录、导入初始版本、设置合适的参数，等等。如果组织级并未提供合适的工具，则需考虑工具的选择、购买、安装、设置等一系列事宜。

如果公司已经确立了通用的软件配置管理过程，则应该考虑拿来使用。这可能是全盘接受，也可能是在此基础上，根据项目的实际情况，如开发团队规模等因素，进行适当的裁剪和定制。而如果当前项目情况非常特殊，则可能需要部分或全部地为项目特别制定软件配置管理过程。当组织级没有确立或没有完全确立通用的软件配置管理过程时，也会有这样的情况。一般来说，在做软件配置管理计划时，应该同时完成过程方面的准备，或者已经完成对这方面的重要决定。

而在人这一方面，要保证各个软件配置管理相关岗位，已经有合适的人选，并具备了在该项目中完成配置管理方面的工作的能力。他们应该有足够多的经验和能力，如果欠缺，应该考虑进行相关培训。

- **日常工作**

软件配置管理系统日常运行，牵扯到项目团队所有成员，因为所有成员都需要与软件资产打交道，也就需要与软件配置管理打交道。从这个角度讲，不同的角色，所有的团队成员，都是软件配置管理系统的一部分。

而软件配置管理工程师主要的职责是保证软件配置管理系统的正常运行。比如，当工具需要某些维护工作和手动备份工作时，需要做一些相应的工具方面的维护和备份工作；当人员有增减的时候，需要开通和管理相关权限；当使用者遇到的各种麻烦需要解决时，需要回答工具和流程方面的疑难问题，这被称为 Trouble Shooting。另外，还需要为一些软件配置管理的具体事宜、具体案例给出意见或做出决定，比如，具体的某个开发任务，是开分支来支持比较好，还是只使用主干来接受相关变更集的提交好。

当然，在一些研发组织中，软件配置管理工程师同时兼任集成工程师，也承担集成工程师的职责。

- **监控、调整与改进**

软件配置管理系统的运行，既需要参与者的深刻理解和自觉执行，也需要某种形式的监控。这首先是因为，每个团队成员都有其关注的焦点，比如开发人员集中精力于编写代码，测试人员集中精力于发现程序中的问题，项目经理集中精力于项目的按时完工，并让客户满意。而遵循软件配置管理规定，并不是他们关注的焦点。其次还因为，每个人都有失误的时候，都有犯错的时候，如果这些不被及时指出和纠正，就会被效仿。而监控的目的就在于，及时发现问题，及时发现没有遵守纪律的问题，并敦促改正。

但是，即使团队成员严格遵守项目中软件配置管理的各项规定，让软件配置管理系统严格按照规定运行，也仍然有可能出问题。因为，随着项目的演进，项目所在的环境，项目对软件配置管理的要求，都在不断变化。比如，变更请求管理越是到项目后期越是严格。因此，项目的软件配置管理策略、方法，需要随着项目的进展进行调整。这是第一类调整。

而有些调整是因为另外的原因，是因为在制定计划时，不能准确地估计到项目对软件配置管理的需求；不能准确地估计到在这个项目中，各项软件配置管理工作应该怎么做，应该做到什么程度。比如，在项目之初，可能认为在这

个项目中为试探性的工作开分支是没有必要的，因为预计没有比较大的试探性工作。但是项目实际运行中，确实遇到了比较大的试探性工作。这时候就需要进行调整。这是第二类调整。

随着对软件配置管理的认识不断提高，会对流程进行调整；随着新的软件配置管理工具的选型、购买、安装设置的完成，会对项目所用工具进行调整。这类以提升为主要特征的调整，通常被称为改进。改进往往具有组织级的意义，最终是在全公司实施。详见第 19.2 节的介绍。

- **收尾**

在产品发布、项目收尾阶段，软件配置管理同样需要进行收尾工作。这主要包括两个方面，软件资产的整理和归档工作、软件配置管理本身的总结和共享工作。

软件资产的保护，在项目收尾时进入关键时刻。这个时候保存的软件资产，是业已成型的软件资产，是最有价值的资产。同时，这个时候的软件资产，是上市软件产品对应的软件资产。这个断面，在将来很有可能被用来作为维护和继续开发的基础。而另一方面，这个时候的软件资产，如果不及时进行整理和归档工作，又容易被渐渐遗忘，最终丢失。随着硬件设施的迁移，随着其他的变迁，先是很难找到它到哪里去了，进而被从物理介质上消除，再也无从恢复。

软件资产的整理和归档，包括对源代码、对安装包、对文档的妥善保存；包括对各个历史版本，特别是发布产品对应的历史版本的妥善保存。这可能通过某种备份来完成；也可能是转移存储地点，从活跃的经常被存取的存储地点，转移到不活跃但更为安全的存储地点。

而对构建方法的整理和归档，对于环境的整理和归档，也同样重要。否则，在将来，有可能即使复现了当时的源代码，却不能生成对应的安装包；于是源代码资产也失去了价值。

对变更请求等相关数据也应该进行整理和归档，保证将来仍可查阅。

这些是对软件资产的整理和归档工作，而对软件配置管理本身，同样存在着整理总结的工作。因为，软件配置管理本身也是一种资产，被称做过程资产，这也是企业的重要资产。

在项目即将结束时，是一个很好的机会，对整个项目过程中的软件配置管理工作进行总结。其中的经验和教训，值得日后其他项目参考。在日后其他项目的起始阶段，制订软件配置管理工作计划时，就要考虑以往的经验和教训。

在整个项目过程中，可能对软件配置管理流程进行了调整和改进。对这些要进行分析，看是否有可能加入到组织一级的软件配置管理流程中，从而使日后其他项目受益。在工具方面，如果引入了新的软件配置管理工具，或对已有工具进行了定制或二次开发，也要考虑，这些成果能否上升为组织一级，供日后其他项目使用。

19.4　平衡集权与自治

集权与自治是个古老的话题。对集权的追捧大有人在，而对自治的支持，也毫不逊色。说点公道话，其实集权有集权的好，自治有自治的好，不可以一味强调其中一端，不可偏废。古希腊的城邦政体固然好，自由、民主、独立，然而缺乏城邦之间的统一领导，邦分崩离析而不能守[2]，最终灭亡。而柏拉图的理想国，终归只是理想国。近现代，对高度集权的经济体制的尝试，也终归只是尝试。

对于企业运作，对于软件研发，也是同样的道理。企业，特别是高新科技企业，越来越意识到，完全的从上至下的科层体制和命令式管理是行不通的，因为基层才更了解实际情况，更知道具体问题该如何处理，更富于创造性。当

[2] 出自《论语 · 季氏》，相关的成语是祸起萧墙。

然，一盘散沙式的管理也同样行不通，这会带来大量的重复劳动，恶性的内部竞争，甚至是工作目标本身的分散。

具体到软件配置管理，也是一样。让每个产品开发团队都独自去搭建一套版本控制系统，独自购买、安装、设置和备份，听起来不合理。同样，如果让企业内所有的产品研发，不论是一个人两周完成的内部工具的研发，还是一千人两年完成的大型项目的研发，都遵循同样的软件配置管理过程，这听起来问题更大。

为什么要集权？对于软件配置管理来讲，最重要的原因有两点。首先，集权通过共享而节约了资源，节约了劳动，说白了就是省钱。就像刚才举的例子，版本控制系统的搭建和维护，是一件有一定工作量的事情，是一件有一定技术含量的事情。在公司一级统一做，只花了一份钱，只雇请了一个相应的人才。而如果几十个产品，各个都要自己搞，那就要花更多的钱，并且雇请更多的人[3]。

其次，车同辙，书同文，有利于沟通和交流，也方便资源的重新配置。比如，如果因为公司业务需要，一个开发人员从一个产品开发团队调到另一个产品开发团队，结果他发现，本来比较相似的产品，比较相似的产品研发，却使用了截然不同的软件配置管理工具，使用了截然不同的软件配置管理流程。于是，他不得不花大量的精力，来学习和适应。相反，如果这两个产品的研发，使用了相同或相似的软件配置管理工具和流程，这个开发人员就能把更多的精力放在软件开发本身上了。

然而，集权和统一并不是绝对的。各个产品的研发，有它特定的背景，因而有对软件配置管理的不同程度的要求或不同方面的要求。就好像人人都要穿鞋，这是一致的，然而，各有各的尺码，不可强求，不可削足适履[4]。至于具体如何组织和运作，要根据具体情况掌握。

---

[3] 尽管这并不是增长几十倍这样的线性增长。

[4] 出自《淮南子·说林训》。《淮南子》，西汉皇室贵族淮南王刘安招致宾客，在其主持下，集体创作而成。此书采百家之长，内容庞杂，近乎一部“先汉学术史”。

以工具为例，一般来说，一个企业的软件配置管理工具，应该统一购买、统一安装、统一设置、统一备份、统一维护和升级。总之，统一管理。这样能够降低成本，也减少企业为管理所牵扯的精力。当使用的是开源的工具或免费的工具时，虽然没有购买的问题，但仍有统一取得、统一安装等问题，仍然应该统一管理。这同样是出于成本考虑。

有的时候，必须对取得的工具加以改造，以适应本企业的需要。这可能是通过在工具之上进行某种封装来完成，也可能在源代码级进行修改，甚至是完全自己写一个工具。这些也尽量在公司一级统一来完成，不要每个产品研发项目都尝试单搞一套。

另一方面，不同类型的、不同背景的研发项目，对工具可能会有不同的要求。为此，可以考虑通过对工具的设置来完成，通过对工具的不同使用方法来完成。比如，有的项目使用分支，以引入多层集成，有的项目则不需要。

如果需求差距比较远，也可以考虑使用不同的工具，比如，二三十人的小项目，而且所有人员都坐在一起开发，用 Subversion 就不错，但上千人的大项目，分布在多个研发站点的大项目，就应该考虑使用 Git 这样的工具。即便如此，仍应该尽量减少工具的品种，比如，仅使用两种版本控制工具，一种用于内部工具开发项目，一种用于产品项目。

流程方面也是类似的道理。几乎可以肯定的是，各产品的研发，不应该采用完全不同的软件配置管理流程和规范。同样，几乎可以肯定的是，各产品的研发不应该采用完全相同的软件配置管理流程和规范。那么，中庸之道在哪里？

如果几个产品的研发，其背景基本相同，对软件配置管理的需求基本相同，那么它们应该遵循相似或相同的流程和规范。从这一点出发进行考虑，应该在企业级建立标准的软件配置管理流程和规范。

对于具体产品的研发，可能需要对标准流程和规范做一些更改：可能是更细化其中的一些内容，可能是加强其中的一些内容，也可能是有所删减。这些更改，统称为流程的**裁剪**（Tailor）[5]。比如，如果只有一个产品，使用了开源项目的成果，作为一个静态链接库，那么，如何管理这个静态链接库的版本等相关的方法、规范和流程，就应该写在这个产品研发本身的软件配置管理计划文档或者这个产品研发本身的流程文档里。而如果有越来越多的产品，或者预计有越来越多的产品会使用这个第三方的静态链接库，那就应该把对这个静态链接库的管理，放到标准流程和规范中。

如果某个产品的研发特点，与标准流程的适用范围非常不同，那么，裁剪可能就不再是一个有效的方法。与其把一件大人衣服改成一件洋娃娃穿的衣服，还不如为洋娃娃单做一件衣服。

对这种情况，适宜为这个产品本身做一套流程规范，并且考虑相关工具的配合。而如果在将来要研发若干相似的产品，这套流程规范适用于或基本适用于一系列产品的研发，那么，就把这套流程规范上升为企业的标准流程规范。也就是说，这时，企业里有不止一套标准流程规范。各产品，根据自己的情况，选择相应的流程规范，并考虑进行少量裁剪。

19.5 管理众多的项目

当公司的产品、组件比较多了之后，管理起来，会遇到一些新的挑战。比如，存储目录结构方面的挑战：公司可能拥有多台专用服务器，每台服务器上有多个版本库，而每一个版本库里，拥有多个产品或组件的源代码。因此需要为此进行规划：设置合理的层次，进行适当的分组，并依此确定各产品、组件的存储位置。记录并遵循这样的规划。如果这样的规划需要调整，就要重新评估，重新设计方案、讨论和批准。

[5] 注意，不是裁减。

不仅是源代码存储有存储结构规划的问题，对文档的存储、对安装包的存储、对静态链接库的存储，也有类似的问题。

不仅是存储和版本控制中存在着这样的问题，变更管理，特别是变更请求管理系统中，也有类似的问题。例如，如果一个缺陷跟踪系统里，拥有数十个甚至数百个产品或组件，那么每次填写缺陷记录或查找缺陷记录时，都需要先在系统中，从这数十个甚至数百个产品或组件中，定位到这个产品或组件。这将会是件相当繁琐的事情。如果在定位过程中，引入某种层次关系，就会大大地提高效率。而对于某个特定的团队或特定的人，一般只关心其中少数几个产品或组件，如果为此设置这几个产品或组件的某种快捷方式，会进一步提高工作效率。

要注意，同一资产，其命名应该统一。一个源代码组件，有一个固定的名称。在版本库中，这个名称反映为存储目录名；在基线和分支中，如果需要组件的名称，那么也用这个名称；在变更请求管理工具里，也要用这个名称；生成的运行组件，还是要用这个名称。除了名称外，版本号也要对应和统一，这个我们以前也提过。

19.6　软件配置管理团队的组织结构

一般来说，研发软件的企业，只要企业不是太小，都会有软件配置管理团队，再不济，也会有一两名软件配置管理人员。他们的职责是什么？

他们要负责对软件配置管理工具和环境的设置和维护。这包括工具的选型、工具的购买或取得、工具的安装、工具的设置、工具的定制和二次开发、工具的升级、工具的数据备份、工具的权限管理、工具的性能调优等。总之，是围绕软件配置管理工具的管理工作[6]。

[6] 在一些研发组织中，版本控制工具等软件配置管理工具的安装和维护工作，由 IT 部门而非软件配置管理团队完成。还有一些组织中，工具的二次开发工作，由工具团队而非配置管理团队完成。总

除了操作和管理工具外，他们还要负责与工具相关的培训。教会每个工程师如何使用这些工具。而当大家在使用过程中遇到问题的时候，也会去请教他们，让他们帮忙解决。

在企业级，设定专人来负责这些事情，比每个项目自己管理要好。软件配置管理工具，特别是大型商用的软件配置管理工具，管理和维护起来，可不是那么容易的。术业有专攻，还是让专人来负责整个企业这方面的事情比较好。

除了工具需要管理，软件配置管理的流程、规范和方法也需要管理。这包括制定流程规范，也包括监督执行，还包括不断地调整和改进。请注意，这里所说的流程规范，包括标准的流程规范，也包括裁剪，还包括个别特定的规范。这些事情，谁来做呢？

通常，公司一级的软件配置管理人员，应该发挥重要作用，因为它更专业，更能照顾全局。而在具体项目中，项目中的配置管理人员也应该发挥作用，因为项目成员更了解本项目的实际情况。一般的标准流程规范，应当由公司一级的软件配置管理人员负责起草、发布并持续改进，并征询各方意见。而具体项目中对标准流程规范的裁剪，公司一级的软件配置管理人员可以只是参与。

企业的软件配置管理人员，应该对员工进行软件配置管理方面的流程培训，仅有工具培训是不够的。要教会每个工程师，如何在这些流程下规范地工作。而当大家在工作过程中遇到问题的时候，也要去请教软件配置管理人员。

对流程和规范的监督执行，也就是常说的流程审计，可以由软件配置管理人员进行，也可以由质量保证部门进行。而在各个项目研发团队内部，可能还有自查。

与公司级的软件配置管理人员相对应的，是项目或产品一级的软件配置管理人员。他们负责项目或产品整个生命周期中的软件配置管理，从制定软件配

之，实际情况，特别是分工较细的大型研发组织中的实际情况，可能与这里介绍的不同。

置管理计划开始。他们还可能会负责产品之间的协调工作，当然这也可能是由企业级的软件配置管理人员完成的。

事实上，项目或产品一级的软件配置管理人员，也有可能在行政上归属于企业级的软件配置管理团队。或者，项目级的软件配置管理人员与企业级的软件配置管理人员，仅仅是角色的划分，实际上是同一个人。

集成工程师，包括组件级的集成工程师和系统级的集成工程师，他们的主要职责是集成，以及与集成相关的一些操作。他们可能由软件配置管理人员兼任。

然而，配置管理人员绝不要把集成、发布等工作当作自己职责的全部。配置管理人员最根本的任务，是建立、维护、调整和改进配置管理解决方案，包括流程、工具、人，保证配置管理系统的运行和不断调优。其中非常重要的一点是，设置正确的、符合企业特定环境特定需求的配置管理策略。让我们为此共同努力！

结语　提倡务实的精神

下面是个小故事：

我家门口有个“蒸功夫”包子铺。我常去那儿买包子吃包子。有一次，老板跟我聊起来，问我是干什么工作的。我说，软件配置管理，就是一帮人一起编程序，我帮忙组织组织、规范规范。老板说，哦，我觉得我们这儿的管理也有点儿乱，要么你给出出主意？

以下是我出的主意。

“首先，先要考虑包子的原材料有哪些，做包子需要哪些炊具灶具，需要煤气罐还是木柴，所有这些，都要登记造册。”

我看到老板点了点头，于是继续往下说。

“为了简化起见，我们拿做一笼包子来考虑：步骤一，标记一下各原料。比如，面一坨。我们就要填表，表上先给这坨面起个名字，比如 蒸功夫-北京-39-面-2007-09-19-18:01:22。表示“蒸功夫”北京第 39 号店，现在刚和好的一坨面。嗯嗯，这样不够专业，现在都兴用版本号，比如蒸功夫-北京-39-面-3.1.9.build3721。不过，如果是这样的话，需要对版本号有严格的规定，什么时候升第一位，什么时候升第二位，什么时候升第三位。第四位数字是 Build Number，也就是构建号，此位只升不降，只要全国“蒸功夫”店新和出一坨面，这一位就要加 1。也就是说，每次你和好面，需要给“蒸功夫”总部打个电话，申请一个新的构建号。这个是很重要的，不然面和面之间就容易混了。全国唯一的号，这多清晰!”

老板欲言又止，因为我用坚定的目光看着他。我接着说，

“不仅面是这样，馅儿也是这样。油啊盐啊，要记录下是哪个牌子的，用的是哪一袋哪一包。我们要保持原料的可跟踪关系。里面放的猪肉馅儿也一样，要跟踪到是哪头猪的肉。呃，买猪肉的时候，您把猪的编号抄下来没有？现在猪有编号没有？猪的族谱有没有？”

老板摇了摇头，“现在还没有，但据说政府打算……”

“没关系，你买肉的时候，在每块儿肉上编一个号、再在一张表格里写上这块肉多重，从哪个摊位进的，几时几分进的、再让肉贩子签个字。这样，出了问题你就没责任了，是肉贩子没建好跟踪关系。不过，你一定要建好肉块儿和肉馅儿之间的跟踪关系，把肉馅按每个包子的分量分成小份，并记录下每小份对应于哪块肉。”

“包子皮也一样，每个包子皮都要编号，不是那一坨面编了号就可以的。对了，刚才忘说了，对那一坨面，还应该有个属性，可以取一些枚举值，比如，已和好、已醒好、已擀好、已包好、还有作废——对应于掉地下的情况。随着面的形态的变化和质量的变化，这个属性的值也在变化，专业的说法是，Promote Baseline!”

“另外，还有一个复合基线的概念。每个包子皮有个编号，每个包子馅儿有个编号，那么，当它们包在一起之后，就应该有一个总体的新的编号，包子的编号。并且，这个编号与包子皮的编号和包子馅儿的编号之间，应该有明确的对应关系。也就是说，包子作为一个子系统，它的编号与它的组件之间的编号应该有一定的对应关系，这种关系应该被记录。在这里，包子的编号，就是复合基线……”

我刚开始说的时候，老板满脸疑惑，时不时问一两句。后来就不问了。瞠目结舌。老板的小儿子后来也凑过来听听。那小子是个愣头青，素质比较低。他听了一阵，回身抄起一把铁锹，骂骂咧咧的，向我扇过来。我夺路而逃……

边逃边遗憾，刚谈了谈配置识别，我还有配置控制、状态统计和配置审计没有说呢……配置管理工作真是不好做啊……

故事当然是杜撰的。相信读者看了会会心一笑。

软件配置管理来不得教条，不是照搬某个标准，照搬某一本书就可以的。同样，作为本书，仅仅是描绘了软件配置管理的基本概念、基本原理和一些常常比较好用的方法，以此勾勒出软件配置管理的框架，可以作为你思考问题、组织工作的起点，供你借鉴参考，然而切不可照搬。

不同的开发背景、不同规模的团队、不同的协作方式等，都会对施行哪些管理、具体的施行方法、以什么样的力度施行等产生不小的影响。本书的内容，需要你结合你所在的研发环境的具体情况，灵活务实地应用。

如果说有一条根本性的原则，那就是，要经常思考这个问题：你为你所在的研发团队制定的软件配置管理策略和方法，是不是真正有价值？也就是说，投入和耗费是不是会有很好的回报和收益？

预祝你成功！

附录 A　相关工具索引

A.1　版本控制工具

[Subversion]

在本文首次出现的位置：第 2.3 节

在本文首次出现时的介绍：

Subversion，缩写为 SVN。SVN 是当前最受欢迎的开源的版本控制工具之一。它继承了它的前辈 CVS（Concurrent Version System）的几乎所有的优点，去掉了 CVS 的几乎所有的弱点，还增加了很多新功能。对于中小型团队，它很值得考虑使用。而对于正在使用 VSS 和 CVS 的广大用户，它可以作为未来版本控制系统升级的目标。

官方网站：http://subversion.tigris.org/

[Git]

在本文首次出现的位置：2.5 节

在本文首次出现时的介绍：

Git 之父就是 Linux 之父 Linus Torvalds。2005 年，他为管理 Linux 内核的开发而研制了 Git。由于 Git 功能强、速度快，越来越多的项目特别是大型开源项目开始采用 Git，这包括 Perl、Gnome、Android，等等。Git 是分布式版本控制系统的典型代表。

官方网站：http://git-scm.com/

[ClearCase]

在本文首次出现的位置：第 2.3 节

在本文首次出现时的介绍：

ClearCase，有时候缩写为 CC。ClearCase 是 IBM Rational 出品的大型商用软件配置管理工具，其核心是版本控制。尽管能够听到对 ClearCase 的很多抱怨，比如昂贵、复杂、不好用，但它仍然是收费的版本控制软件中，市场份额最大的。它有两个版本：Base ClearCase 和 ClearCase UCM。

Base ClearCase 向你提供的是文件、目录、版本、标签、分支、触发器和链接等“裸露”的环境。在此基础上，你可以进行比较自由的设置和二次开发，以满足你实际项目的需要。它的优点是灵活。

ClearCase UCM 是开箱即用（Out of the Box）的，它提供了基于 Base ClearCase 的一套不错的封装。你只需要做一些简单的设置，就可以在实际项目中直接使用了。它的一些性能指标也比 Base ClearCase 优异。

官方网站：http://www.ibm.com/software/awdtools/clearcase/

[Visual SourceSafe]

在本文首次出现的位置：2.3 节

在本文首次出现时的介绍：

Visual SourceSafe，缩写为 VSS，是微软公司出品的一款“古老的”版本控制工具。它默认的方法就是第一种方法，串行的方法。在修改前，锁定相应的文件，以免别人同时修改。在修改后，解锁。通过设置，它也支持下面的第二种方法。它支持基本的版本控制功能，如版本保存、打标签等。另外，它还号称支持分支（见第 7 章）等高级功能，不过，不太好用。总之，它是一款初级的版本控制工具，并且微软已不再维护它。

官方网站：http://msdn.microsoft.com/en-us/vstudio/aa700907.aspx

[其他版本控制工具]

在本文首次出现的位置：2.3 节

在本文首次出现时的介绍：

市面上还有至少几十种版本控制工具，或者包含了版本控制功能的工具。除上文已给出的外，比较出名的还有 Perforce、StarTeam、AccuRev、TFS（Team Foundation Server）、RTC（Rational Team Concert）、Telelogic Synergy 等收费产品和 Mercurial 等开源免费工具。

A.2　变更请求管理工具

[Bugzilla]

在本文首次出现的位置：9.1 节

在本文首次出现时的介绍：

这是开源世界当前最为流行的缺陷跟踪软件，准确地讲，是变更请求管理软件。类似的开源软件还有 Mantis 和 Trac。

官方网站：http://www.bugzilla.org/

[BugFree]

在本文首次出现的位置：9.1 节

在本文首次出现时的介绍：

这是中国人自己的缺陷跟踪软件，开源，简单，用户广泛。最近它成为禅道项目管理软件的一部分。

官方网站：http://www.zentao.net/

[ClearQuest]

在本文首次出现的位置：9.1 节

在本文首次出现时的介绍：

IBM Rational 公司出品。功能强大同时价格不菲的商业版缺陷跟踪软件。类似的商用软件还有 Telelogic Change 等。

官方网站：http://www.ibm.com/software/awdtools/clearquest/

[JIRA]

在本文首次出现的位置：9.1 节

在本文首次出现时的介绍：

这几年 JIRA 变得越来越热门。开源社区可以免费开源地使用它。而如果你的公司用它，那就需要付很少的一点钱，然后也可以得到它的源代码，并二次开发。以上是它的商业策略。在功能方面，大家对它的评价是“功能全面，界面友好，安装简单，配置灵活，权限管理以及可扩展性方面都十分出色”。事实上，它不仅是缺陷跟踪工具，也是项目管理工具。

官方网站：http://www.atlassian.com/software/jira/

A.3 持续集成与构建管理相关工具

[CruiseControl]

在本文首次出现的位置：6.3 节

在本文首次出现时的介绍：

开源世界里的老牌持续集成工具。提供了一个持续集成的框架。一般来说，稍作设置和补充，就能够实现上文所提到的自动化的持续集成。缩写 CC，有时候指的是它，有时候指的是 ClearCase[1]。

官方网站：http://cruisecontrol.sourceforge.net/

[Jenkins/Hudson]

在本文首次出现的位置：6.3 节

在本文首次出现时的介绍：

持续集成工具，新秀，开源。前些年叫 Hudson，最近因为 Oracle 跳出来索要商标权，所以只好改名叫 Jenkins。Jenkins/Hudson 和 CruiseControl 相比，操作界面更友好，配置更容易，运行更稳定。越来越多的项目开始使用 Jenkins/Hudson。

官方网站：http://jenkins-ci.org/　http://hudson-ci.org/

[ElectricCommander]

在本文首次出现的位置：6.3 节

在本文首次出现时的介绍：

Electric Cloud 公司出品。就是出产神奇的构建加速器 ElectricAccelerator 的那个公司。

官方网站：http://www.electric-cloud.com/products/electriccommander.php

[1] 当然，CC 还可能指别的事情。在外企里，你大概会经常听到类似于“把那封邮件 CC 给张老三”之类的中英夹杂的话。CC 还有可能表示立方厘米、云计算等等。

[BuildForge]

在本文首次出现的位置：6.3 节

在本文首次出现时的介绍：

IBM Rational 出品。

官方网站：http://www.ibm.com/software/awdtools/buildforge

[Clearmake]

在本文首次出现的位置：5.3 节

在本文首次出现时的介绍：

Clearmake 是 ClearCase 自带的。使用 Clearmake 而非 make，有助于提高增量构建的可靠性。Clearmake 根据源文件的版本信息，而非根据源文件的修改时间，来判断该文件是否应该重新编译。在考虑使用它的时候，你一定要实测一下它在你特定的应用场景中的工作效率。

官方网站：

http://publib.boulder.ibm.com/infocenter/cchelp/v7r0m0/index.jsp?topic=/com.ibm.rational.clearcase.tutorial.doc/a_clearmake.htm

[ElectricAccelerator]

在本文首次出现的位置：5.4 节

在本文首次出现时的介绍：

Electric Cloud 公司最耀眼的产品。通过一系列加速技术，它能够大幅度提高构建的效率，比如提高 20 倍。当然，它的价格也不菲。

官方网站：http://www.electric-cloud.com/products/electricaccelerator.php

A.4　同行评审工具

[Gerrit]

在本文首次出现的位置：4.2 节

在本文首次出现时的介绍：

Gerrit 是一款开源的同行评审工具，与 Git 相集成。它“拦截”开发人员提交的变更集，只有通过了同行评审才会“放行”。Gerrit 通过网页展现变更集；评审人员可以把评审意见写到对应的代码行附近，方便直观。

官方网站：http://code.google.com/p/gerrit/

A.5　文档管理工具

[SharePoint]

在本文首次出现的位置：8.1 节

在本文首次出现时的介绍：

微软的 SharePoint，至少是一个文档管理系统。此外，它还扩展了 Exchange Server；还提供了可扩展的协同工作解决方案；还支持网站建设，是高度可定制的；还扩展了 InfoPath，一种用于集成表格管理和数据传输的桌面应用程序技术……

官方网站：http://sharepoint.microsoft.com/

[MediaWiki]

在本文首次出现的位置：8.3 节

在本文首次出现时的介绍：

MediaWiki 全球最著名的开源 Wiki 引擎。它是维基百科全书的系统软件，并有大量其他应用的实例。它对中文的支持也很不错。

官方网站：http://www.mediawiki.org/

[其他 Wiki 系统]

在本文首次出现的位置：8.3 节

在本文首次出现时的介绍：

其他 Wiki 系统还有很多，比如老牌的 Twiki，小巧的 DocuWiki 和 Wikka Wiki。它们各有优缺点。你在网上能找到很多它们之间相互比较的帖子。

A.6 安装与部署管理工具

[RPM]

在本文首次出现的位置：18.1 节

在本文首次出现时的介绍：

RPM，全称 RPM Package Manager，是 Linux 世界里使用最广泛的安装包管理系统。 类似的还有 Debian 上的 dpkg，等等。

官方网站：http://www.rpm.org/

[PyPI]

在本文首次出现的位置：18.1 节

在本文首次出现时的介绍：

PyPI，全称 Python Package Index，是用 Python 语言写成的软件的官方存储库。它与 CPAN 有点像。CPAN 存储 Perl 语言写成的软件。

官方网站：http://pypi.python.org/pypi

[Android Market]

在本文首次出现的位置：18.1 节

在本文首次出现时的介绍：

Android Market 是一个由 Google 为 Android 系统用户创建的服务，允许安装了 Android 系统的手机和平板电脑用户从 Android Market 浏览和下载一些应用程序。用户可以购买或免费试用这些应用程序。苹果公司的 App store 也类似。

官方网站：https://market.android.com/

[Windows Installer 和 InstallShield]

在本文首次出现的位置：18.2 节

在本文首次出现时的介绍：

Windows Installer 是 MS Windows 上最流行的安装工具，能安装以 msi 为扩展名的安装包。然而制作这样的安装包可不是一件容易的事，幸亏有了 InstallShield 来帮忙。

官方网站：

http://msdn.microsoft.com/en-us/library/cc185688%28VS.85%29.aspx

http://www.flexerasoftware.com/products/installshield.htm

[Inno Setup 和 NSIS]

在本文首次出现的位置：18.2 节

在本文首次出现时的介绍：

在 MS Windows 上，如果你想免费制作安装包的话，可以考虑上述两个开源软件。还不错。

官方网站：

http://www.jrsoftware.org/isinfo.php　http://nsis.sourceforge.net/

参考书目[1]

Bellagio David E，Milligan Tom J．软件配置管理策略与 IBM Rational ClearCase．2 版．王海鹏，译．北京：人民邮电出版社，2006。

Kenefick Sean．软件配置管理．王海涛，沈火林，译．北京：清华大学出版社，2004。

Bays Michael E．软件发布方法．章柏幸，杨作兴，译．北京：清华大学出版社，2002。

Berczuk Stephen P，Appleton Brad．软件配置管理模式．黄明成，译．北京：中国电力出版社，2004。

Mason Mike．版本控制之道——使用 Subversion．陶文，译．2 版．北京：电子工业出版社，2007。

Travis Swicegood．版本控制之道——使用 Git．董越，付昭伟等，译．北京：电子工业出版社，2010。

Kacobson Ivar，Griss Martin，Jonsson Patrik．软件复用结构、过程和组织．韩柯，译．北京：机械工业出版社，2003。

Boehm Barry，Turner Richard．平衡敏捷与规范．邓辉，孙明译．北京：清华大学出版社，2005。

Kroll Per，Macisaac Bruce．敏捷与秩序——RUP 最佳实践．朱剑平等，译．北京：清华大学出版社，2006。

Vu John．过程改进与 CMM 实践问答．赵悦，郝海静，译．北京：人民邮电出版社，2006。

[1] 显然，并非所有参考书目都被列出了。比如，《法律的概念》（[英]H.L.A.Hart 著）就没有被列出。该书第一章的标题为“恼人不休的问题”。在这一点上，该书被本书参考。

在线资源

国际

CM Crossroads，http://www.cmcrossroads.com/，国外最大的软件配置管理社区。

Branching Patterns，http://www.cmcrossroads.com/bradapp/acme/branching/，涵盖了几乎所有的分支使用模式。

Continuous Integration，http://martinfowler.com/articles/continuousIntegration.html，持续集成的经典论文。内有链接指向中译版。

Wikipedia，http://en.wikipedia.org/wiki/Main_Page，尽管列在这儿有点傻，但它确实是个学习配置管理相关概念的好地方。

国内

SCMLife，http://www.scmlife.com/，目前国内最大最活跃的软件配置管理社区，也涉及软件工程其他领域。

MSUP（麦思博），http://www.msup.com.cn/，一家面向软件研发团队的培训咨询机构，提供包括软件配置管理在内的培训咨询服务。

UML 软件工程组织，http://www.uml.org.cn/，一个软件工程网站，包括软件配置管理相关内容。

Subversion 中文站，http://www.subversion.org.cn/，Subversion 的一个中文站点，也涉及软件配置管理其他内容及软件工程其他领域。

我用 Subversion，http://www.iusesvn.com，Subversion 的一个中文站点，也涉及软件配置管理其他内容及软件工程其他领域。

CM Thinking，http://www.cmthinking.com/，一个 CM 方面的个人网站，内容挺有意思。

书网互动

感谢您选购 SCMLife（www.SCMLife.com）系列精品图书，本书是 SCMLife 系列精品图书中的一本。您购买的不仅是这本书，更享有 SCMLife 的贴心线上服务。

首先，您可以在 SCMLife“谈书论道”频道与作者、读者进行交流，与众多高手一起点评精彩段落，与作者做近距离的沟通，欣赏更多业内精彩书评和推荐，一切尽在“谈书论道”（http://bbs.scmlife.com/forum-72-1.html）。

其次，您可以将在配置管理实践中遇到的困惑、问题在 SCMLife 上发贴，会有朋友在第一时间为你答疑解惑。

再次，您可以参加线下俱乐部定期聚会，直接与厂商、技术专家面对面沟通，汲取经验，结识更多志同道合的业内朋友。

总之，SCMLife 作为国内最大、最全、最专业的配置管理社区，将始终坚持协作、分享、共赢的核心思想，与您一起构筑纯技术、原创社区。